Títol:	Validación de esquemas conceptuales especificados en UML y OCL
Volum:	1/1
Alumne:	Guillermo Lubary Fleta
Directora:	Anna Queralt Calafat
Departament:	Llenguatges i Sistemes Informàtics
Data:	23 de Gener de 2008

DADES DEL PROJECTE

Títol del Projecte: Validació d'esquemes conceptuals en UML i OCL

Nom de l'estudiant: Guillermo Lubary Fleta
Titulació: Enginyeria Informàtica
Crèdits: 37,5
Directora: Anna Queralt Calafat
Departament: Llenguatges i Sistemes Informàtics

MEMBRES DEL TRIBUNAL *(nom i signatura)*

President: Ernest Teniente López

Vocal: Sebastià Xambó Descamps

Secretària: Anna Queralt Calafat

QUALIFICACIÓ

Qualificació numèrica:
Qualificació descriptiva:

Data:

AGRADECIMIENTOS

Creo que es importante dar las gracias por la ayuda y el apoyo que he recibido, sin los cuales, difícilmente, este proyecto podría haber llegado a buen puerto.

En primer lugar, agradezco a Anna Queralt y Ernest Teniente su enorme paciencia y la orientación que constantemente me proporcionaron.

También me gustaría agradecer al equipo de desarrolladores del proyecto EinaGMC, Albert Tort y Elena Planas, su tiempo y la valiosa información que me facilitaron y que me permitió comprender el núcleo de la librería de EinaGMC y la librería del XMIConverner. También agradezco a Antonio Villegas su trabajo al adaptar el Parser OCL para este proyecto.

Por último, he de dar las gracias a Virginia y a mi familia, especialmente a mi madre, que me ayudó a optimizar el tiempo de respuesta de la herramienta. Su apoyo fue crucial cuando las cosas iban bien. Y sobre todo cuando las cosas no iban tan bien.

CONTENIDOS

1. INTRODUCCIÓN 11
1.1. LA MOTIVACIÓN Y EL CONTEXTO DEL PROYECTO 11
1.2. LOS OBJETIVOS DEL PROYECTO 12
1.3. UNA APROXIMACIÓN AL MÉTODO QUERALT 12
1.4. LAS FASES DE DESARROLLO DEL PROYECTO 13
1.5. LA TECNOLOGÍA UTILIZADA Y EL ENTORNO DE TRABAJO 14
1.6. PLANIFICACIÓN DEL PROYECTO 15
1.6.1. Estimación del coste temporal y diagrama de Gantt 16
1.6.2. Estimación del coste económico 17
2. LENGUAJES DE ESPECIFICACIÓN 19
2.1. UML 19
2.1.1. Clase 20
2.1.2. Atributos 20
2.1.3. Generalización 20
2.1.4. Asociaciones 21
2.2. OCL 22
2.2.1. Estructura de un invariante 22
2.2.2. Navegación 23
2.2.3. Operaciones 23
2.3. LÓGICA 24
2.3.1. Constante 24
2.3.2. Variable 25
2.3.3. Término 25
2.3.4. Predicado 25
2.3.5. Átomo 25
2.3.6. Literal 26
2.3.7. Regla o cláusula normal 26

3. CARGANDO UN MODELO UML/OCL 29
3.1. ¿QUÉ ES XMI? 29
3.2. LA DISYUNTIVA ENTE EINAGMC Y ECLIPSE 30
3.3. CODIFICANDO UN MODELO UML EN XMI 31
3.4. PARSEANDO RESTRICCIONES TEXTUALES EN OCL 33
3.5. IMPORTANDO UN MODELO CODIFICADO EN XMI 34

4. TRADUCCIÓN DE UML/OCL A LÓGICA 35
4.1. TRADUCCIÓN DE UML A LÓGICA 36
4.1.1. Traducción de clases, atributos y asociaciones 36
4.1.2. Traducción de las restricciones gráficas implícitas en el modelo 36
4.1.2.1. Restricciones referentes a los OIDs 37
4.1.2.2. Restricciones referentes a las jerarquías 37
4.1.2.3. Restricciones referentes a las associaciones 38
4.1.2.4. Restricciones referentes a las cardinalidades 38
4.2. TRADUCCIÓN DE OCL A LÓGICA 41
4.2.1. Simplificación de las operaciones 41
4.2.2. Traducción de invariantes 43
4.2.2.1. Traducción de navegaciones 43
4.2.2.2. Traducción de selecciones 44
4.2.2.3. La traducción completa de un invariante OCL 46

5. GENERACIÓN DEL GRAFO DE DEPENDENCIAS 49
5.1. VIOLADORES POTENCIALES Y REPARADORES 50
5.2. NODOS Y ARCOS DEL MULTIGRAFO DIRIGIDO 52
5.3. ARCOS SUPERFLUOS 54

6. DETECCIÓN Y ANÁLISIS DE CICLOS EN EL GRAFO 57
6.1. TEOREMA 1 58
6.2. TEOREMA 2 58
6.3. TEOREMA 3 60
6.4. EL GRAFO DE DEPENDENCIAS RESULTANTE 61

7. VALIDACIÓN DEL ESQUEMA 63
7.1. SATISFACCIÓN DE LAS PROPIEDADES DESEADAS (GOAL) 63
7.2. EL MANTENIMIENTO DE LA INTEGRIDAD DEL MODELO 65
7.2.1. Ejemplo de validación de una query 67
8. LOS ENTRESIJOS DE LA HERRAMIENTA AURUS 69
8.1. ESPECIFICACIÓN DE LA HERRAMIENTA 69
8.1.1. El metamodelo lógico 69
8.1.2. El grafo de dependencias 70
8.1.3. Validación del esquema 71
8.1.4. Casos de uso 72
8.2. DISEÑO E IMPLEMENTACIÓN 74
8.2.1. Visión general 74
8.2.2. Traducción a lógica 75
8.2.2.1. Traducción de UML a lógica 77
8.2.2.2. Traducción de OCL a lógica 78
8.2.3. Generación y análisis del grafo de dependencias 80
8.2.3.1. Generación del grafo 81
8.2.3.2. Detección y análisis de los ciclos en el grafo 82
8.2.4. Validación del esquema 84
8.2.4.1. La ventaja de almacenar estados de ejecución 85
8.2.4.2. El problema de la explosión combinatoria 85
8.2.4.3. Detección de violaciones relacionadas con los OIDs 87
9. EJEMPLO 89
10. CONCLUSIONES Y PROBLEMAS ABIERTOS 97
10.1. SOBRE LA HERRAMIENTA AURUS 97
10.2. SOBRE EL COSTE REAL DEL DESARROLLO DE LA HERRAMIENTA 97
10.3. SOBRE EL PROYECTO EN GENERAL 100
12. BIBLIOGRAFÍA Y REFERENCIAS 103
13. ANEXO: EJEMPLO DE CONVERSIÓN UML - XMI 105

1. INTRODUCCIÓN

1.1. La motivación y el contexto del proyecto

Este proyecto se concibió como complemento y demostración empírica para las ideas expuestas por la profesora Anna Queralt en la tesis doctoral cuyo nombre ha tomado prestado este Proyecto de Fin de Carrera: “Validación de esquemas conceptuales en UML y OCL”. Dicha tesis, dirigida por Ernest Teniente, describe un método para verificar si un modelo conceptual especificado en UML y OCL es correcto.

Podemos entender un modelo o esquema conceptual como un mapa de conceptos (clases) y sus relaciones (asociaciones) que intenta representar algo que existe en la realidad. El lenguaje UML (Universal Modeling Language) está considerado un estándar para especificar modelos conceptuales y se usa en todo el mundo. Se trata de un lenguaje gráfico que a menudo se apoya en otros lenguajes textuales, como en OCL (Object Constraint Language), para expresar información adicional.

Así pues, la motivación de este PFC es precisamente desarrollar de una herramienta software que implementa el método descrito en la tesis doctoral de Anna Queralt. Esta aplicación, que ha recibido el nombre de Aurus, se enmarca dentro del proyecto EinaGMC [1], una iniciativa del Grupo de investigación en Modelización Conceptual de la UPC. El propósito de EinaGMC es conseguir un entorno tecnológico de desarrollo que permita trabajar con esquemas conceptuales especificados en UML y OCL. Por lo tanto, la herramienta descrita en este PFC se nutre de este trabajo, como se explicará con detalle, y al mismo tiempo consigue aportar su grano de arena.

1.2. Los objetivos del proyecto

El propósito principal de este PFC es el desarrollo de una herramienta, Aurus, que permita validar automáticamente esquemas conceptuales en UML y OCL, es decir, comprobar si un cierto fragmento de la realidad se ha conseguido plasmar correctamente.

En primer lugar, la herramienta determina si un modelo dado es semánticamente correcto, comprobando, por ejemplo, si existen contradicciones o redundancias, si todas las clases y asociaciones puedan tener instancias, etc.

Pero, además, Aurus es capaz de responder en un tiempo razonable las preguntas que le plantee el usuario, que tiene un punto de vista subjetivo sobre el modelo que ha propuesto. Este usuario será, típicamente, un analista de sistemas software que desea saber si el esquema que ha diseñado es correcto y representa adecuadamente la realidad tal como él tenía en mente. Nos referimos a preguntas que deberán ser planteadas en forma de proposiciones lógicas. Aurus responderá si estas sentencias son o no son satisfactibles bajo las restricciones que impone el modelo (y el propio usuario con sus preguntas). Comparando la respuesta obtenida con la esperada, el analista sabrá si va por buen camino o debe corregir su planteamiento.

1.3. Una aproximación al método Queralt

Actualmente existen otras herramientas cuya finalidad es exactamente ésta: verificar la integridad de un esquema o modelo conceptual. Sin ir más lejos, en esta misma universidad, en el mismo departamento que ha impulsado este proyecto, encontramos un buen ejemplo: la Eina SVT, basada en el método CQC [4]. No obstante, el método CQC está pensado para esquemas de bases de datos, no en UML y OCL.

Entonces, ¿qué es lo que aporta este nuevo validador? Tomando como base otros métodos existentes, el método Queralt constituye una versión simplificada y más eficiente, que da solución a algunos de los problemas de los que adolecen aquéllos.

Un problema del método CQC es que es posible que el proceso de validación no acabe nunca. Este método consiste en la construcción de un ejemplo que satisfaga las propiedades y respete

las restricciones de un modelo dado, es decir, se debe conseguir una instanciación válida para las clases y asociaciones de éste. Se trata de un proceso complicado, en el que se pueden caer en bucles infinitos con facilidad. Es lo que se conoce formalmente como un problema no decidible, lo que significa que no existe un algoritmo capaz de resolverlo aun disponiendo de espacio y tiempo ilimitados. El origen de esta indecidibilidad de encuentra en el enorme poder expresivo de los lenguajes de especificación (en especial, OCL), que pueden provocar que el esquema necesite instancias infinitas.

El nuevo método que implementa la herramienta evita estos bucles infinitos. Realiza un análisis previo del esquema antes de proceder a su validación. De este modo, se asegura de que va a ser capaz de dar una respuesta a las preguntas del usuario, si es que es posible darla.

Una explicación detallada del mecanismo de *reasoning* del método Queralt se puede encontrar en el artículo *Decidable Reasoning in UML Schemas with Constraints* [3] así como en el capítulo 7 de esta memoria.

1.4. Las fases de desarrollo del proyecto

Debido a la envergadura del proyecto, se definieron cinco fases para el desarrollo de la herramienta Aurus, una para cada una de las tareas fundamentales que ésta lleva a cabo.

Fase 1: Cargando un modelo UML/OCL. En primer lugar, se debe cargar en la memoria el esquema conceptual especificado en UML y OCL que se desea validar. Se trata de un proceso que no parece albergar mucha dificultad. Sin embargo, es un proceso muy complejo y fundamental para empezar a manipular el modelo. Afortunadamente, en este proyecto se ha podido aprovechar las librerías implementadas por el grupo de trabajo GMC [1].

Fase 2: Traducción de UML/OCL a lógica. El siguiente paso es traducir el esquema, que está definido con los lenguajes de especificación UML y OCL, y obtener las proposiciones lógicas equivalentes. Para ello, se aplica el sistema de traducción descrito en el artículo *Reasoning on UML Class Diagrams with OCL Constraints* [2]. Esta traducción es necesaria para la fase 3.

Fase 3: Construcción del grafo de dependencias. Se trata de ordenar las restricciones que establece el modelo introducido, en sus dependencias. Este grafo será de gran utilidad para el análisis de la decidibilidad del problema de la validación y para el propio algoritmo de validación.

Fase 4: Detección y análisis de ciclos en el grafo. Antes de proceder a la validación del modelo, debemos asegurar que una vez empezado, el proceso acabará en un tiempo finito. Para ello debemos, en primer lugar, detectar los ciclos o bucles del grafo de dependencias. Una vez detectados, debemos analizarlos para saber si se trata de ciclos finitos o infinitos.

Fase 5: Validación del modelo. Si el modelo carece de ciclos infinitos que puedan obstaculizar el algoritmo de validación, podemos proceder al proceso de validación. Como hemos dicho, básicamente consiste en la construcción de un ejemplo satisfactible, para lo cual se ejecuta un algoritmo de *backtracking* que prueba todas las posibles combinaciones hasta que éstas se agotan.

1.5. La tecnología utilizada y el entorno de trabajo

El validador de esquemas conceptuales especificados en UML y OCL que se presenta en este PFC ha sido implementado en el lenguaje de programación Java, lenguaje imperativo y orientado a objetos. Se podrían enumerar muchos motivos según los cuales escoger Java ha sido una buena idea (y también algunos motivos para lo contrario), pero la ventaja principal es que las librerías de EinaGMC, tanto su núcleo como sus aplicaciones adicionales, fueron implementadas en Java. Por lo tanto, la única manera de aprovechar este trabajo ha sido utilizar el mismo lenguaje.

Inicialmente, no obstante, se estuvo dudando en cambiar el planteamiento. Se estuvo dudando en abandonar el apoyo que suponía trabajar dentro del proyecto EinaGMC y se llegó a plantear utilizar los recursos que ofrecía Eclipse.

Finalmente, Eclipse sólo acabó imponiéndose como entorno de desarrollo para la elaboración de la herramienta. La necesidad de disponer de unas librerías fiables, cuyas funcionalidades, ventajas y limitaciones estuvieran perfectamente determinadas, así como disponer de un soporte cercano y transparente, personificado en el equipo de desarrolladores del proyecto EinaGMC,

fueron las razones para desarrollar la herramienta en consonancia con EinaGMC. Esta decisión se explica con más detalle en el capítulo 3.3.

1.6. Planificación del proyecto

En todo proyecto es una buena idea realizar una planificación inicial. La planificación ayuda a ser consciente de los recursos de los que se disponen (en este caso, el tiempo), de ver cuánto se ha avanzado en un momento dado, cuánto falta para acabar y es útil para saber si el desarrollo de las etapas del proyecto está yendo al día.

Inevitablemente, durante el transcurso del proyecto, la planificación se reajusta, se corrige y se adapta para poder cumplir con las fechas de entrega, tal vez a costa de ciertas funcionalidades adicionales. Se hará referencia al coste real del proyecto en las conclusiones de la memoria.

La planificación inicial establecía 7 etapas, cuyas primeras 6 corresponden con las 5 fases de desarrollo mencionadas en el capítulo 1.4. La primera fase de traducción se encuentra dividida en dos etapas: traducción UML y traducción OCL. La última etapa se corresponde con la documentación.

La mayoría de etapas de desarrollo cuenta con varias subetapas secuenciales, de manera que para iniciar una se necesita concluir la anterior:

- Estrategia: Hace referencia a la comprensión del problema y el esbozo de la solución.
- Especificación: Las clases y relaciones necesarias para dar forma a la solución.
- Diseño: Elaboración de los componentes con sus métodos, considerando la tecnología.
- Implementación: Materialización de las clases y las funciones diseñadas en código Java.
- Pruebas: Testeo de la funcionalidad de cada componente mediante ejemplos concretos.

Ya al principio del proyecto se sabía que, de una manera u otra, no sería necesario especificar ni diseñar un nuevo mecanismo para cargar modelos, si no que se pensaba aprovechar la tecnología existente. Por esta razón, la primera etapa del proyecto (ver diagrama de Gantt) se salta estas subetapas y se centra en la implementación, es decir, en la adaptación de la tecnología a nuestra disposición.

1.6.1. Estimación del coste temporal y diagrama de Gantt

Inicialmente se evaluó el coste temporal del proyecto en 600 horas. Concretamente, 500 para el desarrollo de la herramienta y 100 para la documentación del proyecto. A continuación se muestra la distribución de horas de trabajo y periodos temporales:

Tareas	Inicio	Fin	Horas
Cargando un modelo	1-4-08	17-4-08	102
Estrategia	1-4-08	8-4-08	8
Implementación	9-4-08	11-4-08	6
Creación ejemplos	12-4-08	16-4-08	10
Pruebas	17-4-08	17-4-08	2
Traducción UML	18-4-08	24-5-08	76
Estrategia	18-4-08	27-4-08	10
Especificación	28-4-08	1-5-08	8
Diseño	2-5-08	6-5-08	10
Implementación	7-5-08	22-5-08	32
Pruebas	17-5-08	24-5-08	16
Traducción OCL	25-5-08	21-7-08	137
Estrategia	25-5-08	29-5-08	5
Especificación	30-5-08	1-6-08	6
Diseño	2-6-08	9-6-08	16
Implementación	10-6-08	19-7-08	80
Pruebas	7-7-08	21-7-08	30
Construcción grafo	22-7-08	24-8-08	63
Estrategia	22-7-08	28-7-08	7
Especificación	29-7-08	2-8-08	10
Diseño	4-8-08	6-8-08	6
Implementación	7-8-08	21-8-08	30
Pruebas	20-8-08	24-8-08	10

Tareas	Inicio	Fin	Horas
Análisis ciclos	25-8-08	16-9-08	51
Estrategia	25-8-08	29-8-08	5
Especificación	30-8-08	1-9-08	6
Diseño	2-9-08	6-9-08	10
Implementación	7-9-08	16-9-08	20
Pruebas	13-9-08	17-9-08	10
Validación esquema	18-9-08	26-10-08	74
Estrategia	18-9-08	27-9-08	10
Especificación	28-9-08	29-9-08	4
Diseño	30-9-08	9-10-08	20
Implementación	10-10-08	24-10-08	30
Pruebas	22-10-08	26-10-08	10
Documentación	29-9-08	20-1-09	97
Informe	29-9-08	5-10-08	2
Memoria	23-10-08	21-12-08	75
Presentación	8-1-09	20-1-09	20
			600

Situando el inicio del proyecto en abril de 2008 la lectura del proyecto a mediados de enero de 2009, se planificó desarrollar la herramienta entre abril y noviembre de 2008. Se reservaban, de este modo, las últimas semanas para la elaboración de la memoria y la preparación de la defensa delante del tribunal.

Figura 1: Diagrama de Gantt de la evolución real del proyecto

1.6.2. Estimación del coste económico

Suponiendo que la retribución media aproximada de un analista-programador sea de 35 € / hora, y que sean necesarias 500 horas para desarrollar el producto, el coste de este proyecto ascendería a 500*35 = 17.500 €.

2. LENGUAJES DE ESPECIFICACIÓN

Tal como decíamos al principio, podemos entender un modelo o esquema conceptual como un mapa de conceptos (clases) y sus relaciones (asociaciones) que intenta representar un pedazo de realidad.

Podemos modelizar casi cualquier cosa, cualquier empresa, cualquier jerarquía familiar, cualquier situación cotidiana… Por ejemplo, podemos modelizar personas, ciudades y la relación que hay entre personas y ciudades.

El lenguaje UML (Universal Modeling Language) está considerado un estándar para especificar modelos conceptuales y se usa en todo el mundo. Se trata de un lenguaje gráfico que a menudo se apoya en otros lenguajes textuales, como OCL (Object Constraint Language) para expresar información adicional.

A continuación se comentan únicamente las características más destacables de ambos lenguajes. Su definición formal completa se puede encontrar en [5] y [6]. Además, en el punto 2.3 se explican los conceptos sobre la lógica de primer orden necesarios para comprender los próximos capítulos de esta memoria.

2.1. UML

El lenguaje universal para el modelado (UML), estandarizado por OMG (Object Modeling Group) y actualmente en su versión 2.0, sirve para especificar, visualizar y documentar esquemas de sistemas de software orientado a objetos. No se trata de un método de desarrollo, es decir, no sirve para determinar cómo diseñar el sistema, sino que simplemente proporciona una representación gráfica para visualizar del diseño y hacerlo más accesible a terceros.

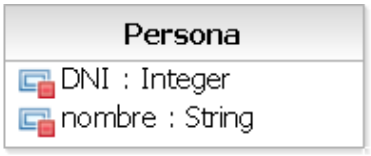
UML se compone de muchos elementos de esquematización que representan las diferentes partes de un sistema de software. En este PFC nos centraremos en los diagramas de clases.

Los diagramas de clases muestran las diferentes clases que componen un sistema y cómo se relacionan unas con otras. A continuación veremos algunos de sus elementos más característicos.

2.1.1. Clase

Una clase define los atributos y los métodos de una serie de objetos. Todos los objetos o instancias de esta clase tienen las mismas propiedades o atributos, con un valor diferente (o no) cada uno de ellos.

En UML, las clases están representadas por cajas rectangulares, con el nombre de la clase, y , adicionalmente, atributos y operaciones. De todas maneras, en este proyecto nos centramos únicamente en la parte estructural de los esquemas, razón por la que no consideraremos las operaciones.



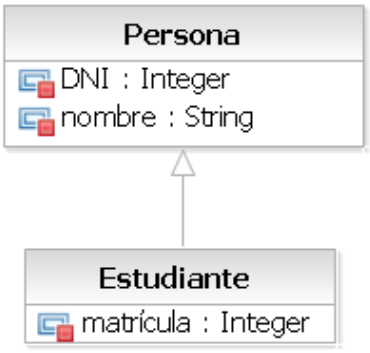
2.1.2. Atributos

En UML, los atributos (o propiedades) se muestran al menos con su nombre, y también pueden mostrar su tipo, valor inicial y otras características. En el ejemplo anterior, Persona tiene dos atributos: DNI, de tipo integer, y nombre, de tipo String.

2.1.3. Generalización

Las generalizaciones o herencias permiten que una clase (general o padre) comparta sus atributos y operaciones con una clase derivada de ella (específica o hija). La clase derivada puede alterar/modificar algunos de ellos, así como añadir más atributos y operaciones propias.

En UML, una asociación de *generalización* entre dos clases, coloca a estas dos (o más clases) en una jerarquía que representa el concepto de herencia.

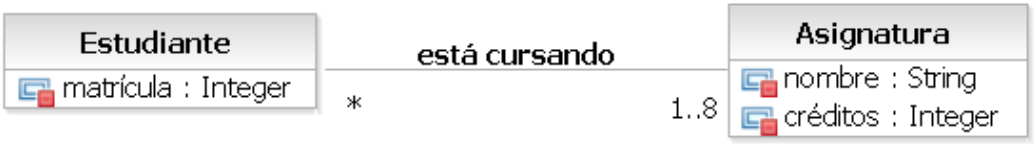


2.1.4. Asociaciones

Una asociación representa una relación entre N clases, y aporta la semántica común y la estructura de muchos tipos de conexiones entre objetos.

Las asociaciones pueden tener un nombre que especifica el propósito de la asociación y pueden ser unidireccionales o bidireccionales (indicando el tipo de navegabilidad entre ellos). Cada extremo de la asociación también tiene un valor de multiplicidad, que indica cuántos objetos de ese lado de la asociación están relacionados con un objeto del extremo contrario. Además, cada extremo puede tener un nombre de rol, precisamente para aclarar su papel en la asociación.

En UML, las asociaciones se representan por medio de líneas que conectan las clases participantes en la relación, y también pueden mostrar el papel y la multiplicidad de cada uno de los participantes. La multiplicidad se muestra como un rango [mín...máx] de valores no negativos.



2.2. OCL

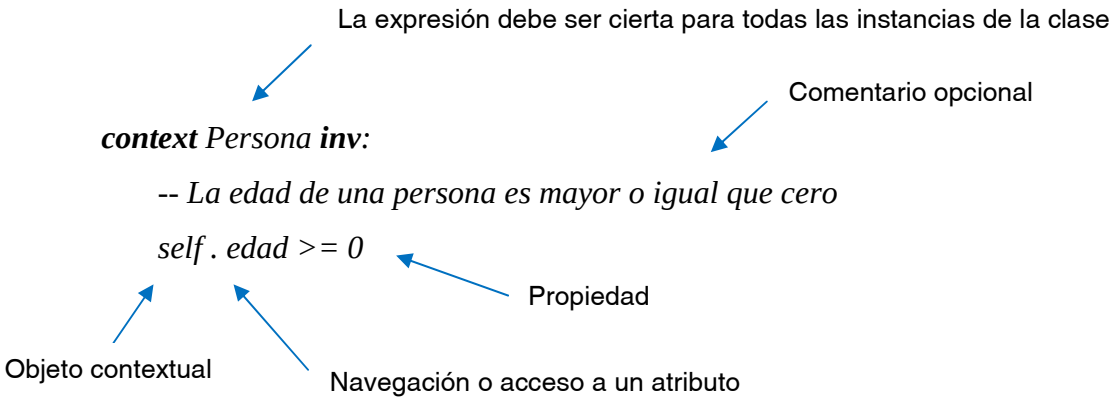
OCL (Object Constraint Language), actualmente en su versión 2.0, fue adoptado en 2003 por el grupo OMG como parte de UML 2.0. Se trata de un lenguaje para la descripción formal de expresiones en los modelos UML. Tales expresiones pueden representar invariantes, precondiciones, postcondiciones, inicializaciones, reglas de derivación, etc.

En este proyecto nos centramos en los invariantes, que permiten especificar restricciones textuales que completan la información gráfica de un modelo conceptual en notación UML. La siguiente explicación está basada en [7].

2.2.1. Estructura de un invariante

Todo invariante está encabezado por el objeto de contexto, es decir, la instancia alrededor de la cual estará redactada la expresión del invariante. En realidad, el invariante define una propiedad sobre este objeto contextual. Para definir dicha propiedad, OCL accede a sus atributos, navega a través de las asociaciones entre clases y establece condiciones o compara elementos.

Veamos un ejemplo:



2.2.2. Navegación

A partir del objeto contextual se puede navegar a través de las asociaciones del modelo para referirnos a otros objetos y/o sus atributos. Se navega utilizando el operador binario ".", escribiendo el objeto de origen y el objeto o atributo de destino. Éste último se especifica con el nombre de atributo, si es un atributo. Si es un objeto o colección de objetos, se usa el nombre de rol en el extremo de la asociación correspondiente con la clase del objeto u objetos; o, en caso de carecer de nombre de rol, se utiliza el propio nombre de la clase.

Si el destino de una navegación es una colección de objetos, esta colección puede ser un conjunto propiamente dicho, un multiconjunto (con elementos repetidos) o una secuencia (donde los elementos están ordenados).

2.2.3. Operaciones

Sobre las colecciones se pueden realizar varias operaciones: select, collect, forAll, exists, isUnique, size. Dado que en este PFC se va a utilizar una versión simplificada de OCL, basada en la utilización de la operación select y la operación size, nos centraremos en éstas dos.

- **Select** Especifica un subconjunto de una colección.
  Por ejemplo:
  --Conjunto de personas mayores de 50 años que trabajan en una empresa
  self.empleado->select(edad>50)

- **Size** Retorna la cardinalidad de la colección considerada.
  Por ejemplo:
  --No puede ser que no haya ningún empleado mayor de 50 años
  self.empleado->select(edad>50)->size()>0

2.3. Lógica

Evidentemente, no sólo existen los lenguajes UML y OCL para modelizar la realidad. Sin lugar a dudas, el lenguaje más antiguo y más utilizado a lo largo de los tiempos es la propia lógica. De hecho, la lógica está muy presente de forma implícita en los lenguajes modernos.

Por lo tanto, el tratamiento de las restricciones de integridad y la comprobación de ciertas propiedades de los esquemas conceptuales, que por fuerza Aurus debe realizar, puede ser abordado a través de las herramientas y procesos de la lógica de primer orden.

Así como para UML y para OCL existe una implementación de su metamodelo que Aurus puede aprovechar, no existe un metamodelo que represente la lógica tal como este proyecto requiere. Por esta razón ha habido que implementar un metamodelo propio, basado en el que se expone en [8].

Vamos a resumir algunos conceptos propios de la lógica necesarios para comprender el metamodelo lógico y las expresiones lógicas que se utilizan a lo largo de este PFC, así como nomenclatura propia de este proyecto.

2.3.1. Constante

Una constante es un valor perteneciente a un dominio (por ejemplo, el dominio *Persona*, o el dominio *Ciudad*) que no cambia durante el transcurso del tiempo. Una constante se correspondería con una instancia, por ejemplo, una instancia de *Persona* sería "*John Travolta*". Dicho de otra manera, "*John Travolta*" es un valor fijo que puede tomar Persona, es decir, una constante. Una constante se representa con una letra minúscula $\{a, b, c, a_1, b_1, c_1,...\}$ o bien puede ser $\{0.0, 1.0, 1.2, 5.7,\ldots\}$ si conocemos su valor exacto.

Es importante tener en cuenta que, por simplicidad, en este proyecto todas las constantes serán números, concretamente reales (de tipo *java.util.Float)*, independientemente del dominio al que pertenezcan.

2.3.2. Variable

Una variable es un símbolo que representa un rango de valores perteneciente a un dominio. Se representa con letras mayúsculas $\{A, B,\ldots, X, Y, Z, X_0, Y_1,\ldots\}$.

2.3.3. Término

Un término T es una variable o una constante. Así, $\{A, a, 0.0, \text{"John Travolta"}\}$ es un conjunto de términos.

Una barra horizontal sobre una variable o una constante, representa un conjunto de variables o constantes, respectivamente. Así, si vemos una $\bar{X}$, sabemos que en realidad puede estar refiriéndose a $\{X,Y,Z,\ldots\}$. Y si vemos $\bar{a}$ sabemos que puede significar $\{a,b,c,\ldots\}$.

Por otra parte, el símbolo θ representa una sustitución entre variables y términos (típicamente constantes), es decir, un mapeo de variable/término $\{X_1/T_1,\ldots,X_n/T_n\}$, donde cada variable X_i es única y cada término T_i es diferente de X_i. De esta manera, cuando hallemos una expresión de tipo $E\ \theta$ deberemos interpretarla como E donde sus variables han sido sustituidas por los términos correspondientes.

2.3.4. Predicado

Un predicado es una palabra o letra que identifica un concepto o un dominio. Por ejemplo, *Persona*, *Ciudad* o *PersonaQueViveEnUnaCiudad* serían predicados. Es lo que se correspondería con las clases o asociaciones propias de UML.

2.3.5. Átomo

Sean $T_1, \ldots, T_n$ términos y P un predicado, $P(T_1, \ldots,T_n)$ es un átomo. *Persona("John Travolta")*, *Persona(X)*, *Ciudad(23.2)* o *PersonaQueViveEnUnaCiudad("John Travolta","Los Ángeles")* también son átomos.

2.3.6. Literal

Hay dos tipos de literales:

- Literal ordinal: Es un átomo verdadero o falso, es decir, negado o no. Por ejemplo, $Persona(X)$ y $\neg Persona(X)$ son literales ordinales.
- Literal Built-In: Es una fórmula de la forma: $T_1\ opComp\ T_2$

donde T_1 y T_2 son términos y $opComp$ es un operador de comparación <, >, =, <>,>= ó >=.

2.3.7. Regla o cláusula normal

Una regla (de deducción) o cláusula normal es una fórmula lógica de primer orden de la forma $P(T_1, ..., T_n) \leftarrow L_1 \wedge L_2 \wedge ... \wedge L_n$, donde la parte izquierda, es decir, el *head* $P(T_1, ..., T_n)$, es un átomo que denota una conclusión. El cuerpo de la regla, la parte derecha constituida por $L_1 \wedge L_2 \wedge ... \wedge L_n$, es una conjunción de literales representando una condición.

Se puede leer una regla de la siguiente manera: *"Si todos los literales del cuerpo de la regla son ciertos, entonces el literal del head también lo es".*

Si la regla no tiene cuerpo se dice que su head (el átomo en concreto) es un "hecho", es decir, es cierto siempre. En cambio, si la regla no tiene cabecera se dice que la regla es un objetivo.

En este PFC, este tipo de reglas (o cláusulas normales) que no tienen head serán las más habituales, ya que con este tipo de fórmulas será como representaremos las restricciones de integridad de los modelos que consideremos.

De hecho, cuando nos refiramos a una "regla", usualmente nos referiremos a una "regla sin head", a no ser que explicitemos lo contrario. Y cuando hablemos de "reglas derivadas", estaremos haciendo referencia a reglas con head, porque el átomo de head se encontrará en el cuerpo de otra regla (sin head).

Además, en este contexto, la semántica de la regla será la contraria a la que parece representar. Es decir, leeremos una regla (sin head) de la siguiente manera: “No puede ser que…”. Por ejemplo, la regla :

$$\leftarrow PersonaQueViveEnUnaCiudad(\text{“John Travolta”},\text{”Los Ángeles”}) \wedge \neg Ciudad(\text{“Los Ángeles”})$$

Se puede leer así:

“*No puede ser que John Travolta viva en la ciudad de Los Ángeles si esta ciudad no existe*”.

3. CARGANDO UN MODELO UML/OCL

La primera de las tareas que debe realizar nuestro validador parece trivial: debe aceptar los datos de entrada. Es decir, debe aceptar un modelo o esquema conceptual cuyas restricciones gráficas se encuentren definidas en UML y con restricciones textuales escritas en OCL.

3.1. ¿Qué es XMI?

La primera pregunta que se debe plantear es la siguiente: ¿cómo codificamos un esquema conceptual en UML y OCL? Existen diferentes maneras hacerlo. No obstante, la forma de codificación más habitual y la que mayor aceptación está teniendo últimamente es XMI.

XMI (XML Metadata Interchange) es una especificación estándar basada en el lenguaje de tags XML que permite representar diagramas UML. Se utiliza básicamente para almacenar los diagramas y luego poder recuperarlos. Antes de que existiera XMI, las herramientas de modelado (Poseidon, Rational Rose, ArgoUML…) utilizaban su propio formato, de manera que un diagrama creado en una herramienta dada no podía ser interpretado, ni mucho menos modificado, por otra herramienta de modelado. Esta situación fue la que propició la aparición de un estándar.

Desafortunadamente, la situación no ha parecido mejorar mucho, ya que existen diferentes versiones de XMI, y cada fabricante ha adoptado una u otra, e incluso ha ampliado la versión estándar para aumentar su potencia. Eclipse, por ejemplo, tiene su propia versión de XMI.

Así las cosas, el grupo de trabajo GMC de la UPC decidió crear su propia versión XMI, basada en la 1.1, para utilizar en el proyecto EinaGMC [1].

3.2. La disyuntiva ente EinaGMC y Eclipse

Trabajar con un modelo conceptual especificado en UML y OCL implica tener, en definitiva, una instanciación del metamodelo de UML y OCL. El núcleo de EinaGMC aporta precisamente eso, la posibilidad de cargar el modelo e instanciarlo en clases Java (que representan el metamodelo de UML y OCL) y la posibilidad de manipularlo invocando ciertos métodos y operaciones.

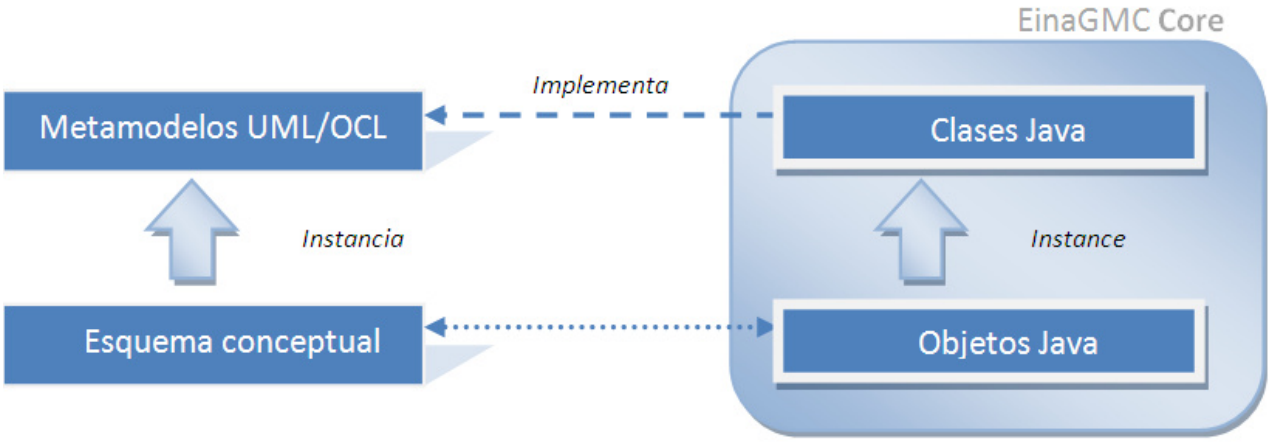


Figura 2: Funcionalidad de API EinaGMC

Sin embargo, después de iniciar el proyecto EinaGMC aparecieron otras iniciativas para crear entornos de trabajo equivalentes. Eclipse, con su Eclipse Modeling Framework [10], concretamente la librería UML2, es uno de los entornos de trabajo que reúne las características adecuadas.

Analizando las características de las APIs de Eclipse, parecía que utilizar este *framework* era la mejor opción. En primer lugar, Eclipse es una comunidad abierta a través de la que sería más fácil hacer visible este trabajo, donde hay mucha gente trabajando y añadiendo nuevas funcionalidades. La librería de Eclipse UML2 basada en EMF implementa el metamodelo de UML. Además, la librería GMF (Graphical Modeling Framework), también de Eclipse, permite representar gráficamente los modelos conceptuales. Es decir, lo tendríamos todo integrado en un mismo entorno.

Entonces, ¿por qué no se ha desarrollado la herramienta Aurus con las librerías de Eclipse? Utilizar las APIs EMF y UML2 tiene algunos inconvenientes. En primer lugar, la construcción de los esquemas. Sí, se pueden definir modelos conceptuales con una potencia bastante elevada.

De una manera u otra, se pueden especificar asociaciones n-arias, por ejemplo. El problema es, precisamente, este "de una manera u otra". Un mismo esquema puede ser creado de maneras diferentes utilizando estas librerías, sin que haya convenciones establecidas. El hecho de que pueda haber documentos XMI diferentes que significan lo mismo es una fuente de complicaciones a la hora de analizar el modelo, ya que tendríamos que prever cada una de las maneras en que el analista puede crear un modelo concreto.

En cambio, utilizando la librería de EinaGMC un mismo modelo sólo se puede crear de una manera: sólo puede haber un documento XMI por modelo conceptual. Las funcionalidades y limitaciones de EinaGMC están perfectamente definidas y documentadas. Además, existía la atractiva ventaja de disponer del soporte de los desarrolladores de EinaGMC en persona.

Finalmente se decidió adaptar Aurus al proyecto EinaGMC, permitiendo aprovechar el trabajo realizado, y, por lo tanto, sólo acepta como entrada diagramas codificados en el formato definido por el grupo de trabajo GMC.

Los problemas de las APIs de Eclipse se deben, sin duda, a que son parte de un proyecto que está en una fase de desarrollo, con funcionalidades todavía inestables. Probablemente en un futuro cercano, no mucho tiempo después de haber terminado este PFC, el trabajo de la comunidad de Eclipse habrá avanzado lo suficiente como para reconsiderar la decisión tomada.

3.3. Codificando un modelo UML en XMI

Como hemos comentado, el formato que la herramienta necesita para poder analizar el esquema conceptual deseado es XMI, en su versión de EinaGMC. El analista puede obtener un documento XMI equivalente al esquema conceptual que ha diseñado en papel de varias maneras.

En primer lugar, se podrían escribir los diagramas UML en formato XMI directamente. No obstante, esta es una tarea muy complicada. Un diagrama que involucrara tres clases con sus asociaciones podría requerir fácilmente más de cien líneas de código.

Otra opción del analista sería implementar él mismo un programa para crear el diagrama deseado utilizando la librería de EinaGMC (código Java) [11]. Es la misma idea del párrafo anterior, añadiendo un intermediario: en lugar de escribir directamente, se invocan operaciones que crean este código XMI. De todas maneras, sigue siendo un trabajo arduo.

Lo natural sería utilizar una herramienta de modelado, como las que hemos comentado anteriormente: Poseidon, Rational Rose, ArgoUML... Sin embargo, no existe ninguna que trabaje con el formato requerido por EinaGMC. La solución a este problema pasa por utilizar la herramienta XMIConverter [12], desarrollada por el propio GMC (concretamente por Elena Planas), que permite transformar la especificación de un diagrama en XMI versión Poseidon a XMI versión EinaGMC y viceversa. Por lo tanto, el analista podría utilizar Poseidon para especificar gráficamente el diagrama de clases, exportarlo como XMI (versión Poseidon) y transformarlo en XMI de EinaGMC mediante el XMIConverter.

Ésta última parece la mejor opción. Sin embargo, hay un problema importante: Poseidon es una herramienta de modelado para crear, visualizar y modificar diagramas UML poco expresiva. Por ejemplo, no se pueden crear asociaciones n-arias: las asociaciones sólo pueden tener dos extremos. Los analistas a menudo necesitan esas asociaciones en sus esquemas. Así que si desean utilizar Poseidon para especificarlos, por necesidad deberán normalizar su diagrama, proceso en el que se obtiene un diagrama equivalente sin asociaciones n-arias.

Existe otra alternativa que permite conservar el esquema original, sin normalizarlo, y no es tan farragoso como escribir XMI a mano o implementar un generador de XMI específico. Se trata de combinar estos procedimientos. Es decir, se diseña con Poseidon todo lo que Poseidon permita, se exporta a XMI de Poseidon, se convierte a XMI de EinaGMC, se carga este modelo parcial en memoria utilizando los métodos de la API de EinaGMC y finalmente se completa invocando las operaciones correspondientes de la misma librería. Éste es el procedimiento que se ha seguido para elaborar los diagramas de clases en este PFC.

Se puede encontrar un ejemplo completo de este procedimiento en el anexo, al final de esta memoria.

3.4. Parseando restricciones textuales en OCL

El lector habrá notado que en los últimos párrafos apenas se ha mencionado OCL. Hemos hablado de diagramas de clases que muestran las restricciones gráficas de un modelo o esquema conceptual, es decir, aquello que se puede especificar mediante UML. Pero seguramente el modelo conceptual necesite restricciones textuales descritas con OCL. Esto quiere decir que la discusión anterior acerca de los formatos XMI, del XMIConverter, de Poseidón… no contempla restricciones textuales en OCL. Y, sin embargo, las necesitamos.

El GMC, concretamente Antonio Villegas Niño, en su proyecto [9] desarrolló un parser de OCL que es capaz de procesar restricciones OCL en formato texto para un modelo dado en formato XMI e incorporarlas al mismo documento XMI. Este parser funciona como una aplicación independiente pero también está integrada en el XMIConverter.

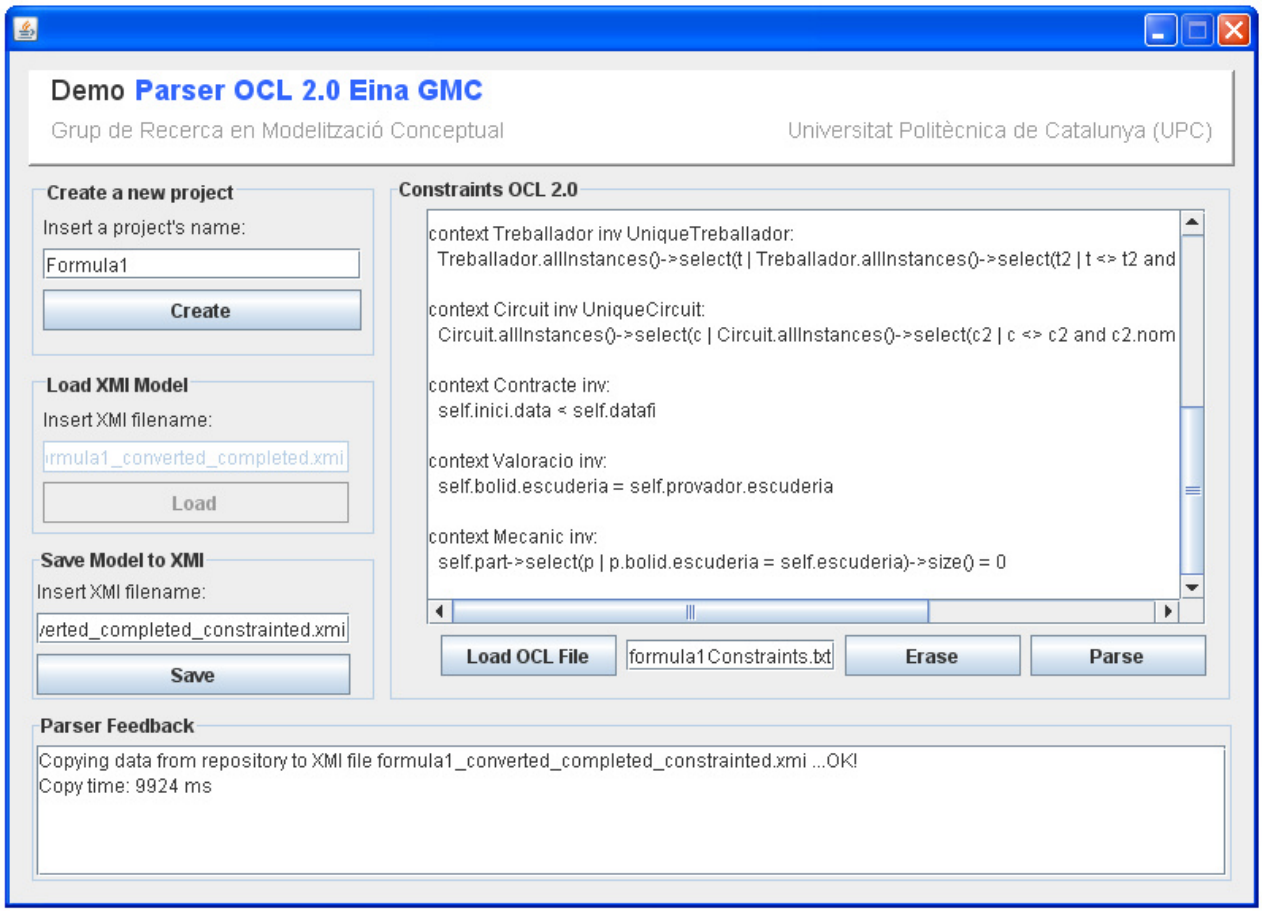


Figura 3: Demo Parser OCL 2.0 EinaGMC

Por lo tanto, el analista tiene dos opciones. La primera opción es tomar el XMI que representa el diagrama de clases es UML, cargarlo en el parser OCL, introducir las restricciones y guardar el nuevo documento XMI. La segunda opción sería, simplemente, añadir las restricciones OCL como comentarios en Poseidon. Al convertir el XMI de Poseidon en XMI de EinaGMC, el XMIConverter las procesa y las inserta en el documento XMI.

3.5. Importando un modelo codificado en XMI

Una vez obtenido el documento XMI, la herramienta debe cargar en la memoria el modelo que representa. Se trata de un proceso que no parece albergar mucha dificultad. Sin embargo, es un proceso muy complejo. Y es fundamental para empezar a manipular el modelo. Afortunadamente, en este proyecto se ha podido aprovechar las librerías implementadas por el grupo de trabajo GMC.

Por lo tanto, este proceso es transparente para la herramienta que se presenta en este PFC. Si se analiza su código, se puede ver que tan sólo son necesarias estas líneas:

```
import project.Project;
Project pro = new Project();
pro.importXMI(input);
```

4. TRADUCCIÓN DE UML/OCL A LÓGICA

Una vez disponemos del modelo conceptual especificado en UML y OCL cargado en memoria, debemos traducirlo a sentencias lógicas de primer orden. La herramienta realiza este proceso automáticamente, siguiendo una serie de reglas y procedimientos que a continuación detallamos. La explicación que sigue recoge el mecanismo de traducción expuesto en el artículo *Reasoning on UML Class Diagrams with OCL Constraints* [2].

Para ilustrar cada caso de traducción, utilizaremos un ejemplo de un modelo conceptual sencillo. Se pueden encontrar ejemplos de traducción más completos en los anexos de esta memoria.

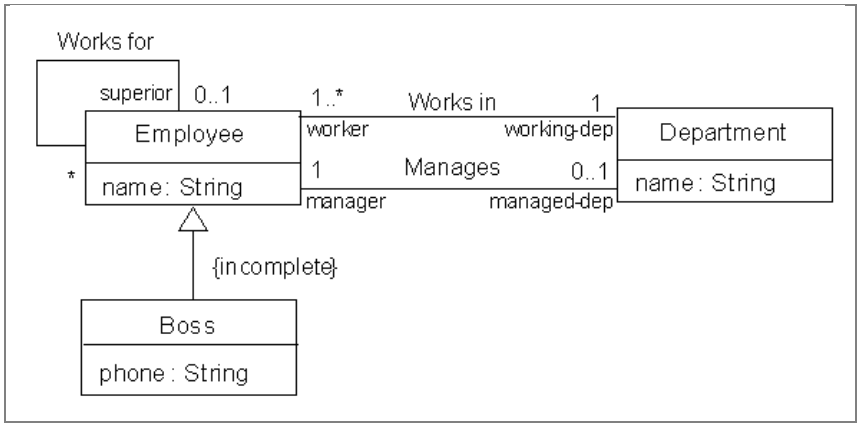


Figura 4: Ejemplo de diagrama de clases UML

Este modelo representa las relaciones entre los departamentos, empleados y jefes de una empresa. En cada departamento hay algunos empleados (al menos uno), cuyo jefe, que también se considera un empleado, trabaja en el mismo. Como máximo cada empleado puede tener un jefe, y cada jefe puede controlar un solo departamento, o ninguno.

En el punto 5.1. analizaremos los mecanismos de traducción desde UML a lógica y en el punto 5.2. nos ocuparemos de las restricciones textuales en OCL. Como veremos, trabajar con éstas últimas es una tarea más complicada.

4.1. Traducción de UML a lógica

4.1.1. Traducción de clases, atributos y asociaciones

En primer lugar, las clases, atributos y asociaciones de UML se representan con predicados. Para cada clase (no asociativa) C se define un predicado unario C, cuyo único término representa el identificador interno del objeto (OID). De esta manera, para el ejemplo de los departamentos, tenemos los siguientes predicados. Por ejemplo, la clase *Employee* se traduciría como *Employee(e)*.

Dada una asociación R entre las clases $C_1,...,C_n$, si R no es una clase asociativa definimos un predicado básico $R(c_1,...,c_n)$. Si *R* es una clase asociativa, se debe añadir un identificador r para la propia clase y el predicado quedaría como $R(r,c_1,...,c_n)$. Aunque no es estrictamente necesario, este identificador resulta útil y permite tratar todas las clases de manera uniforme. Por ejemplo, la asociación *WorksIn* que relaciona empleados con departamentos se traduce como *WorksIn(E,D)*, ya que no es clase asociativa.

Los atributos pueden interpretarse como asociaciones binarias entre una clase *C* y un tipo de datos. Por lo tanto, para cada atributo a_i en *C*, definimos un predicado binario tal que $CA_i(c,a_i)$. El nombre de este tipo de predicados se forma a partir del nombre de la clase y el nombre del atributo para evitar confusiones, ya que podríamos encontrar otro atributo de otra clase con el mismo nombre. Por ejemplo, el atributo *name* de *Employee* se traduce como *EmployeeName(E,N)*.

Todos estos predicados se utilizarán en los literales, dentro de las reglas que representan las restricciones impuestas por el modelo, tanto gráficas (UML) como textuales (OCL).

4.1.2. Traducción de las restricciones gráficas implícitas en el modelo

Por cada restricción gráfica (aunque ocurre igual para las textuales) obtendremos una o varias reglas como puede ser $\leftarrow A(X) \wedge B(Y) \wedge \neg C(Z) \wedge X<>Y$. Debemos recordar que una regla de este estilo se lee como "No puede ser que exista una instancia de A, otra instancia de B y que no haya otra instancia de C, siendo las de A y B diferentes".

4.1.2.1. Restricciones referentes a los OIDs

En primer lugar, debemos garantizar que no van a coexistir dos instancias con el mismo OID. El caso de las instancias de una misma clase ya está garantizado, por la propia semántica de la lógica de primer orden (una instancia $p(a)$ es diferente de una instancia $p(b)$). Sin embargo, tenemos que definir reglas para prevenir la existencia de dos literales de predicados diferentes que usen el mismo OID.

Para cada par de predicados representando clases (siempre que sean clases que no pertenezcan a la misma jerarquía) definimos la regla $\leftarrow C1(X) \wedge C2(X)$. En el modelo de ejemplo tendríamos que definir únicamente la regla $\leftarrow Employee(X) \wedge Department(X)$, ya que sólo esas dos clases pertenecen a diferentes jerarquías.

4.1.2.2. Restricciones referentes a las jerarquías

Las jerarquías de clases necesitan, por otra parte, la definición de un conjunto de restricciones para garantizar que cada instancia de cada subclase es también una instancia de la superclase. La regla genérica sería: $\leftarrow C_{sub\ i}(X) \wedge C_{super}(X);$ y un ejemplo de su aplicación:

$$\leftarrow Employee(X) \wedge \neg Boss(X).$$

Además, es necesario garantizar que una instancia de la superclase no sea una instancia de varias subclases simultáneamente, si es que así está indicado (*disjoint*):

$$\leftarrow C_{sub\ i}(X) \wedge C_{sub\ j}(X)$$

Finalmente, hay que asegurar que, si la jerarquía es completa, una instancia de la superclase pertenezca a una subclase. Para ello necesitamos un predicado derivado y una regla para cada subclase:

$$\leftarrow C_{super}(c) \wedge IsKindOfCsuper(c)$$
$$IsKindOfCsuper(c) \leftarrow Csub\ i(c)$$

4.1.2.3. Restricciones referentes a las associaciones

Otra de las restricciones que en el diagrama de clases UML aparece de forma implícita es el hecho de que los extremos de una asociación, es decir, las clases relacionadas a través de una asociación, existan realmente en el modelo.

Por ello, se define una restricción $\leftarrow R([r,],c_1,...,c_n) \wedge C_i(c_i)$ para cada extremo de cada asociación. En nuestro ejemplo, la asociación *WorksIn* necesita las reglas:

$\leftarrow WorksIn(E,D) \wedge \neg Employee(E)$

$\leftarrow WorksIn(E,D) \wedge \neg Department(D)$

Análogamente, hay que definir restricciones que garanticen que el primer término de un literal que corresponde a un atributo corresponda a una instancia de la clase a la que pertenece el atributo. En el modelo de los departamentos:

$\leftarrow EmployeeName(E,N) \wedge \neg Employee(E)$

$\leftarrow BossPhone(B,P) \wedge \neg Boss(B)$

$\leftarrow DepartmentName(D,N) \wedge \neg Department\ (D)$

Por otra parte, se debe asegurar que no va a existir más de una instancia de una clase asociativa con los mismos extremos. Para cada clase asociativa R que relaciona las clases $c_1,\ldots,c_n$, añadimos la regla:

$\leftarrow R(r_1,c_1...c_m) \wedge R(r_2,c_1...c_m) \wedge r_1 <> r_2$

.

4.1.2.4. Restricciones referentes a las cardinalidades

Finalmente, en lo que se refiere a las cardinalidades de un atributo o de una clase c_i en una asociación R, debemos definir reglas que limiten la cantidad de instancias que puede existir en cada caso. Una cardinalidad está definida genéricamente estableciendo el límite inferior y el límite superior como *min..max* (por ejemplo, 0..1, 1..1, 0..5, 1..*, etc).

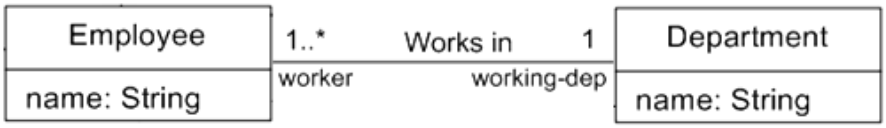
Si *max<** para c_i en R debemos añadir la siguiente restricción, donde vemos que se añaden *max+1* instancias de c_i dentro de *max+1* instancias de R, que mantiene, no obstante, el resto de extremos invariables. Finalmente, se matiza que estas *max+1* instancias de c_i deben ser diferentes entre ellas. Así, queda establecido que no pueden existir max+1 instancias de c_i en la asociación R, lo cual es equivalente a decir que, como máximo, habrá *max* instancias.

$$\leftarrow R([r_1,]c_1,...,c_{i-1},c_{i1},c_{i+1},...,c_n) \wedge ... \wedge$$
$$R([r_{max+1},]c_1,...,c_{i-1},c_{i\ max+1},ci_{+1},...,c_n) \wedge c_{i1}<>c_{i2} \wedge ... \wedge c_{i1}<>c_{i\ max+1} \wedge ... \wedge c_{i\ max}<>c_{i\ max+1}$$

Si *min>0* para c_i en R debemos añadir la siguiente restricción, que, en esencia, es muy similar a la anterior. Lo que sucede es que necesitamos una regla derivada para poder expresar que no puede haber menos de *min* instancias de c_i en R.

$$\leftarrow C_1(c_1) \wedge ... \wedge C_{i-1}(c_{i-1}) \wedge C_{i+1}(c_{i+1}) \wedge ... \wedge C_n(c_n) \wedge \neg MinR(c_1,...,c_{i-1},c_{i+1},...,c_n)$$
$$MinR(c_1,...,c_{i-1},c_{i+1},...,c_n) \leftarrow R([r_1,]c_1,...,c_{i-1},c_{i1},c_{i+1},...,c_n) \wedge ...$$
$$\wedge R([r_{min},]c_1,...,c_{i-1},c_{i\ min},c_{i+1},...,c_n) \wedge$$
$$\wedge c_{i1}<> c_{i2} \wedge ... \wedge c_{i1}<> c_{i\ min} \wedge ... \wedge c_{i\ min-1}<>c_{i\ min}$$

Por ejemplo, consideremos la asociación *WorksIn*. Debemos definir la regla de la izquierda para garantizar que se respeta la cardinalidad inferior de la clase *Employee* (1) y la regla de la derecha para garantizar que se respeta la cardinalidad superior de la clase *Department* (1):



$$\leftarrow Department(D) \wedge \neg MinWorker(D) \qquad \leftarrow WorksIn(E,D1) \wedge WorksIn(E,D2)$$
$$OneWorker(D) \leftarrow WorksIn(E,D) \qquad \wedge D1 <> D2$$

Por lo que respecta a los atributos, si su multiplicidad no está especificada, asumimos que su cardinalidad es 1 y deben tener exactamente una instancia. Por ejemplo, para el atributo *name* de la clase *Employee* necesitaríamos:

$$\leftarrow Employee(E) \wedge \neg MinEmployeeName(E)$$

$MinEmployeeName(E) \leftarrow EmployeeName(E,N)$

$\leftarrow EmployeeName(E,N1) \wedge EmployeeName(E,N2) \wedge N1 <> N2$

4.2. Traducción de OCL a lógica

Vamos a realizar la traducción de restricciones textuales especificadas en OCL a lógica en dos pasos. En primer lugar, simplificaremos las expresiones OCL de manera que sólo aparezcan las operaciones *select* (retorna los elementos de una colección que satisfacen ciertas condiciones) y *size* (retorna la cardinalidad de una colección). El objetivo de esta transformación preliminar es reducir el número de construcciones que debemos traducir, es decir, reducir la sintaxis de OCL. Una vez tengamos la expresión simplificada, procederemos a la traducción a lógica, propiamente dicha.

Del mismo modo que en el punto anterior, ilustraremos la teoría con un ejemplo práctico. Siguiendo con el modelo de los departamentos, cuyo esquema conceptual se encuentra en el punto 5.1., utilizaremos las restricciones textuales en OCL correspondientes.

```
context Employee inv UniqueEmp:
Employee.allInstances()->isUnique(name)

context Department inv UniqueDep:
Department.allInstances()->isUnique(name)

context Department inv ManagerIsWorker:
self.worker->includes(self.manager)

context Department inv ManagerHasNoSuperior:
self.manager.superior->isEmpty()

context Boss inv BossIsManager:
self.managed-dep->notEmpty()

context Boss inv BossHasNoSuperior:
self.superior->isEmpty()

context Boss inv SuperiorOfAllWorkers:
self.employee->includesAll(self.managed-dep.worker)
```

Figura 5: Restricciones textuales en OCL

4.2.1. Simplificación de las operaciones

El primer paso, pues, consiste en transformar una expresión OCL que puede tener múltiples operaciones (*includes, forAll, isUnique, isEmpty*) en otra expresión semánticamente equivalente

que únicamente tenga las operaciones *select* y *size*. En la tabla siguiente se muestran las reglas de transformación correspondientes.

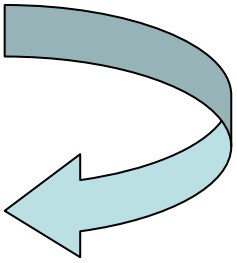
Expresión original	**Expresión equivalente simplificada**
source->includes(obj)	*source->select(e | e=obj)->size()>0*
source->excludes(obj)	*source->select(e | e=obj)->size()=0*
source->includesAll(c)	*c->forall(e| source->includes(e))*
source->excludesAll(c)	*c->forall(e| source->excludes(e))*
source->isEmpty()	*source->size()=0*
source->notEmpty()	*source->size()>0*
source->exists(e | body)	*source->select(e | body)->size()>0*
source->forall(e | body)	*source->select(e | not body)->size()=0*
source->isUnique(e | body)	*source->select(e | source->select(e2 | e <>e2 and e2.body = e.body))->size()=0*
source->one(e | body)	*source->select(e | body)->size()=1*
source->reject(e | body)	*source->select(e | not body)*

Tabla X: Equivalencias de operaciones OCL

Por ejemplo, si consideramos la restricción *ManagerIsWorker* de nuestro modelo, habría que realizar la siguiente transformación:

context** Department **inv** ManagerIsWorker**:
self.worker->includes(self.manager)-> size() > 0

context** Department **inv** ManagerIsWorker**:
self.worker->select(e \|e = self.manager)-> size() > 0



Es importante dejar claro que Aurus, la herramienta que se presenta en este PFC, no implementa esta conversión. Aunque la idea sería que la aplicación aceptara cualquier expresión y que Aurus la simplificara automáticamente, se decidió dejar esta funcionalidad para futuras ampliaciones de Aurus, debido a la envergadura del proyecto. Por lo tanto y por el momento, el proceso de simplificación debe realizarse a mano.

4.2.2. Traducción de invariantes

Una vez simplificados, los invariantes OCL, que son los tipos de expresiones con los que vamos a tratar, tienen la forma

context C ***inv:*** *path-exp->select(e |body)-> size()* opComp k ,

donde C es una clase del esquema conceptual, *path-exp* es una secuencia de navegaciones a través de las asociaciones, opComp es un operador de comparación (<,>,= o <>) y k es un entero mayor o igual que cero.

Hemos eliminado los operadores >= y <=, para poder efectuar el tratamiento que posteriormente se detalla. En cualquier caso, tales expresiones se pueden adaptar si k>0, transformando >=k en >k-1 y <=k en <k+1.

Veremos que la traducción del invariante varía en función del operador de comparación después de *size()*. Posteriormente estudiaremos cómo se traduce la operación selección y, finalmente, daremos una visión global de la traducción. De momento, vamos a ver cómo traducir la navegación definida en *path-exp*.

4.2.2.1. Traducción de navegaciones

Sea $path\text{-}exp = obj.r_1...r_n[.att]$ el camino que empieza en la instancia obj de la clase C, o desde la invocación de la operación allInstances sobre C, sigue a través de los roles $r_1...r_n$ y, opcionalmente, acaba en el acceso a un atributo *att*. Sea $C(obj)$ la traducción a lógica de la clase a la que *obj* pertenece, sean $R_i(obj_{i-1}, obj_i,...)$ los literales correspondientes a la asociación entre los roles r_{i-}, *y* r_i, , y sea C_2 la clase donde el atributo *att* ha sido definido. Esta navegación se traduce a lógica mediante el fragmento siguiente:

$$C(obj) \wedge R_1(obj,obj_1,...) \wedge ... \wedge R_n(obj_{n-1}, obj_n,...) [\wedge C_2(obj_n) \wedge C_2Att(obj_n, att)]$$

Por ejemplo, la navegación *self.worker* de la restricción *ManagerIsWorker* se traduciría como $Department(D) \wedge WorksIn(E,D)$.

4.2.2.2. Traducción de selecciones

Una operación de selección se encuentra habitualmente en esta forma *select(e | body)*, donde *e* se conoce como variable de iteración y *body* no es más que una comparación entre dos elementos que se encuentran al final de un camino de navegación:

$$body = path_1 \text{ opComp } path_2$$

En este caso sencillo, la operación de selección se traduce como

$$Tr\text{-}path(path_1) \wedge Tr\text{-}path(path_2) \wedge obj_1 \text{ opComp } obj_2 ,$$

donde obj_1 y obj_2 son los objetos que se obtienen en el último paso de las navegaciones $path_1$ y $path_2$, respectivamente. Es importante destacar que si alguno de los caminos de navegación es una constante o la variable de iteración (típicamente *e*) no hay que realizar ninguna traducción. Es el caso del siguiente ejemplo.

La expresión *select(e| e=self.manager)* que aparece en la versión simplificada de la restricción *ManagerIsWorker* se traduce como

$$Department(D) \wedge Manages(E2,D) \wedge E=E2,$$

donde se observa que *E*, que es la variable de iteración de la selección y correspondería al $path_1$, no ha requerido traducción, mientras que *self.manager*, correspondiente al $path_2$ se ha convertido en *Department(D)* $\wedge$ *Manages(E2,D)*.

Sin embargo, una operación de selección puede ser más compleja, ya que puede albergar otras operaciones de *select* y/o *size* en su interior. En este caso, la forma general de estas expresiones es así: *body = path-exp->select(e| body)->size()* opComp *k* o, si no hay operación de selección, así: *body = path-exp-> size()* opComp *k*.

A continuación se detalla la traducción de este tipo de selecciones, que varía en función del operador de comparación. En las siguientes expresiones lógicas, *c* representa el objeto de contexto mientras que las variables $e,...,e_m$ que los literales *Aux* necesitan corresponden a las

variables de iteración de todas las operaciones de selección que aparecen en algún momento dentro del *body*.

a) Si opComp es >, $obj.r_1 ... r_{n-1}.r_n$ -> *select(e| body)*-> *size()* > *k* se traduce como:

$$Tr\text{-}path(obj.r_1 ... r_{n-1}) \wedge Tr\text{-}path_1(r_n) \wedge Tr\text{-}select_1(e|\ body)$$
$$\wedge \ldots \wedge Tr\text{-}path_{k+1}(r_n) \wedge Tr\text{-}select_{k+1}(e|\ body)$$

Aquí tenemos una operación de *select* (recordemos que esto está dentro de otro *select*) que está determinando un cierto subconjunto de objetos sobre el conjunto definido por la navegación. Además, se establece como mínimo *k* elementos en este subconjunto.

Expresar esta situación en el lenguaje de la lógica implica repetir la traducción del *subselect* tantas veces como determina *k,* más una (recordemos que ahora estamos dentro del *body* de una operación de selección, no en una regla lógica, ya que esto implicaría establecer la condición inversa).

b) Si opComp es < , $obj.r_1 ... r_{n-1}.r_n$ -> *select(e| body)*-> *size()* < *k* se traduce como:

$$Tr\text{-}path(obj.r_1 ... r_{n-1}) \wedge \neg Aux(e,...,e_m,c)$$

$$Aux(e,...,e_m,c) \leftarrow Tr\text{-}path_1(r_n) \wedge Tr\text{-}select_1(e|\ body)$$
$$\wedge \ldots \wedge Tr\text{-}path_k(r_n) \wedge Tr\text{-}select_k(e|\ body)$$

En este caso nos vemos obligados a utilizar una regla derivada porque no podemos expresar directamente *"debe haber menos de* k *elementos"*. Si repitiéramos la traducción del *subselect k-1* veces, no estaríamos determinando la condición que deseamos, si no que estaríamos forzando que almenos hubiera k-1 elementos. Necesitamos una condición que establezca que hay *k* elementos y luego negarla.

En el siguiente caso se combinan los procedimientos de *a)* y *b).*

c) Si opComp es = , $obj.r_1 ... r_{n-1}.r_n$ -> *select(e| body)*-> *size()* = *k* se traduce como:

$$Tr\text{-}path(obj.r_1 ... r_{n-1}) \wedge Tr\text{-}path_1(r_n) \wedge Tr\text{-}select_1(e|\ body)$$

$$\wedge \ldots \wedge \textit{Tr-path}_k(r_n) \wedge \textit{Tr-select}_k(e|\ body) \wedge \neg \textit{Aux}(e,\ldots,e_m,c)$$

$$\textit{Aux}(e,\ldots,e_m,c) \leftarrow \textit{Tr-path}(obj.r_1 \ldots\ r_{n-1}) \wedge \textit{Tr-path}_1(r_n) \wedge \textit{Tr-select}_1(e|\ body)$$
$$\wedge \ldots \wedge \textit{Tr-path}_{k+1}(r_n) \wedge \textit{Tr-select}_{k+1}(e|\ body)$$

Por ejemplo, consideremos el invariante OCL simplificado de *SuperiorOfAllWorkers*:

context *Boss* ***inv*** *SuperiorOfAllWorkers****:***
self.managed-cat.worker->select(e| self.employee->select(e2 | e2=e)->size()=0)->size()=0

Aplicando la traducción definida en *c)* (la más habitual) para la operación de selección exterior, obtendríamos:

$\textit{Tr-path}(self) \wedge \neg \textit{Aux}(E,B)$
$\textit{Aux}(E,B) \leftarrow \textit{Tr-path}(self) \wedge \textit{Tr-path}(employee) \wedge \textit{Tr-select}(E2|\ E2=E)$

Y si traducimos el select interior obtendríamos:

$Boss(B,P) \wedge \neg Aux(E,B)$
$Aux(E,B) \leftarrow Boss(B,P) \wedge Superior(B, E2) \wedge E2=E$

4.2.2.3. La traducción completa de un invariante OCL

Ahora que ya sabemos cómo traducir ciertos fragmentos del invariante por separado, vamos a ver qué debemos hacer con el invariante completo. Recordemos que un invariante OCL tiene la forma ***context*** *C* ***inv:*** *path-exp->select(e |body)-> size()* opComp *k*, donde *path-exp* puede expresarse como $obj.r_1 \ldots\ r_{n-1}.r_n$.

En función del operador de comparación, definimos la traducción de un invariante en términos de la traducción de la navegación *(Tr-path)* y la traducción de la selección *(Tr-select)* como sigue:

a) Si opComp es <, $obj.r_1 \ldots\ r_{n-1}.r_n$ -> *select(e| body)-> size()* < *k* se traduce como:

$$\leftarrow C(c) \wedge \textit{Tr-path}(obj.r_1 \ldots\ r_{n-1}) \wedge \textit{Tr-path}_1(r_n) \wedge \textit{Tr-select}_1(e|\ body)$$
$$\wedge \ldots \wedge \textit{Tr-path}_k(r_n) \wedge \textit{Tr-select}_k(e|\ body)$$

Al contrario que en el punto 5.2.2.2., para expresar que hay menos de *k* elementos, se define una regla donde aparece la traducción de la selección *k* veces. Así se establece lo que no puede ocurrir en el modelo.

b) Si opComp es > , $obj.r_1 ... r_{n-1}.r_n$ -> *select(e| body)*-> *size()* > *k* se traduce como:

$$\leftarrow C(c) \curlywedge \neg Aux(c)$$

$$Aux(c) \leftarrow Tr\text{-}path(obj.r_1 ... r_{n-1}) \curlywedge Tr\text{-}path_1(r_n) \curlywedge Tr\text{-}select_1(e|\ body)$$

$$\curlywedge \ldots \curlywedge Tr\text{-}path_{k+1}(r_n) \curlywedge Tr\text{-}select_{k+1}(e|\ body)$$

Es ahora, en cambio, cuando nos vemos obligados a utilizar una regla derivada. Ahora lo que no podemos expresar directamente es *"debe haber más de* k *elementos"*. Necesitamos una condición que establezca que hay *k+1* elementos y luego negarla. Al final, lo que tenemos es *"No puede ser que exista (...) y que no tenga al menos* k+1 *elementos"*.

c) Si opComp es = , $obj.r_1 ... r_{n-1}.r_n$ -> *select(e| body)*-> *size()* = *k* se traduce como:

$$\leftarrow C(c) \curlywedge \neg Aux(c)$$

$$Aux(c) \leftarrow Tr\text{-}path(obj.r_1 ... r_{n-1}) \curlywedge Tr\text{-}path_1(r_n) \curlywedge Tr\text{-}select_1(e|\ body)$$

$$\curlywedge \ldots \curlywedge Tr\text{-}path_k(r_n) \curlywedge Tr\text{-}select_k(e|\ body)$$

$$\leftarrow C(c) \curlywedge Tr\text{-}path(obj.r_1 ... r_{n-1}) \curlywedge Tr\text{-}path_1(r_n) \curlywedge Tr\text{-}select_1(e|\ body)$$

$$\curlywedge \ldots \curlywedge Tr\text{-}path_{k+1}(r_n) \curlywedge Tr\text{-}select_{k+1}(e|\ body)$$

d) Si opComp es <> , $obj.r_1 ... r_{n-1}.r_n$ -> *select(e| body)*-> *size()* = *k* se traduce como:

$$\leftarrow C(c) \curlywedge Tr\text{-}path(obj.r_1 ... r_{n-1}) \curlywedge Tr\text{-}path_1(r_n) \curlywedge Tr\text{-}select_1(e|\ body)$$

$$\curlywedge \ldots \curlywedge Tr\text{-}path_k(r_n) \curlywedge Tr\text{-}select_k(e|\ body) \curlywedge \neg Aux(c)$$

$$Aux(c) \leftarrow Tr\text{-}path(obj.r_1 ... r_{n-1}) \curlywedge Tr\text{-}path_1(r_n) \curlywedge Tr\text{-}select_1(e|\ body)$$

$$\curlywedge \ldots \curlywedge Tr\text{-}path_{k+1}(r_n) \curlywedge Tr\text{-}select_{k+1}(e|\ body)$$

Claramente, los casos en que k vale 0 o 1 quedan bastante simplificados (sólo hay que hacer una o dos repeticiones de *Tr-Path* y/o *Tr-Select*) y, de hecho, son los más comunes. Por ejemplo, consideremos la simplificación del invariante *ManagerIsWorker*:

***context** Department **inv:** self.worker -> select(e| e=self.manager)-> size() > 0*

Aplicando la traducción en definida en *b)*, obtenemos:

$\leftarrow Department(D) \curlywedge \neg Aux(D)$

$Aux(D) \leftarrow Tr\text{-}path(self) \curlywedge Tr\text{-}path(worker) \curlywedge Tr\text{-}select(e|\ e=self.manager)$

Si la completamos concretando la traducción de las navegaciones y las selecciones, obtenemos una expresión donde sólo es necesario repetir una sola vez $Tr\text{-}path(worker) \curlywedge Tr\text{-}select(e|\ e=self.manager)$, porque *k* vale 0.

$\leftarrow Department(D) \curlywedge \neg Aux(D)$

$Aux(D) \leftarrow Department(D) \curlywedge WorksIn(E,D) \curlywedge Manages(E2,D) \curlywedge E=E2$

Si *k* hubiera tenido, por ejemplo, el valor 2, el resultado habría sido:

$\leftarrow Department(D) \curlywedge \neg Aux(D)$

$Aux(D) \leftarrow Department(D) \curlywedge WorksIn(E,D) \curlywedge Manages(E2,D) \curlywedge WorksIn(E3,D) \curlywedge$
$Manages(E4,D) \curlywedge WorksIn(E5,D) \curlywedge Manages(E6,D) \curlywedge E<>E3 \curlywedge E<>E4 \curlywedge E<>E5 \curlywedge$
$E<>E6 \curlywedge E3<>E5 \curlywedge E3<>E6 \curlywedge E4<>E6 \curlywedge E<>E5 \curlywedge E<>E6 \curlywedge E=E2 \curlywedge E3=E4 \curlywedge$
$E5=E6$

No obstante, podría ocurrir que el invariante OCL no tuviera ninguna operación. En ese caso, tendría la forma ***context** C **inv:** path-exp* opComp *value*, donde *value* podría representar tanto una constante como otro camino de navegación. La traducción para este tipo de invariantes sería:

$\leftarrow C(c) \curlywedge Tr\text{-}path(path\text{-}exp) \curlywedge Tr\text{-}path(value) \curlywedge obj_1\ invOpComp\ obj_2,$

donde obj_1 y obj_2, son los objetos que se obtienen al final de los caminos de navegación *path-exp* y *value*, y donde *invOpComp* es el operador de comparación inverso. En el caso en que *value* fuera una constante, no habría que traducirla, evidentemente.

5. GENERACIÓN DEL GRAFO DE DEPENDENCIAS

Una vez la herramienta Aurus ha conseguido una versión del modelo conceptual que intenta validar expresada en términos de lógica de primer orden, hay que construir un grafo de dependencias. ¿Dependencias entre qué elementos? Entre cada una de las reglas lógicas (restricciones) obtenidas. ¿Cuáles son estas dependencias y por qué necesitamos un grafo? Para responder estas preguntas vamos a necesitar algunos párrafos más. La idea es que necesitamos ordenar las restricciones para poder efectuar el proceso de validación de forma eficiente. De hecho, antes de eso, necesitaremos saber si existe el riesgo de que tal proceso no acabe jamás y para ello analizaremos el grafo.

En este capítulo, basado en *Decidable Reasoning in UML Schemas with Constraints* [3], se explica cómo generar el grafo de dependencias que se precisa y se comentan algunas cuestiones referentes a su implementación. Ilustraremos la explicación mediante ejemplos referentes a un modelo concreto ya traducido a lógica. Es como sigue:

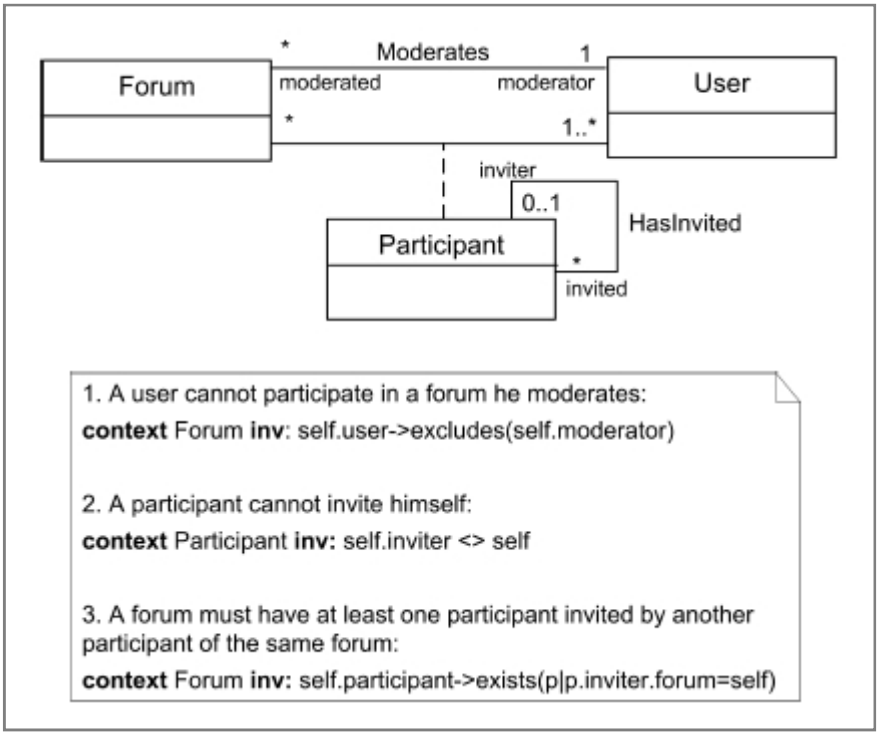


Figura 6: Ejemplo de modelo conceptual

0: $\leftarrow Forum(X) \wedge User(X)$

1: $\leftarrow Forum(X) \wedge Participant(X,F,U)$

2: $\leftarrow User(X) \wedge Participant(X,F,U)$

3: $\leftarrow HasInvited(P0,P1) \wedge \neg IsParticipant(P0)$

$IsParticipant(P0) \leftarrow Participant(P0,F0,U0)$

4: $\leftarrow HasInvited(P0,P1) \wedge \neg IsParticipant(P1)$

$IsParticipant(P1) \leftarrow Participant(P1,F0,U0)$

5: $\leftarrow HasInvited(P1,P0) \wedge HasInvited(P2,P0) \wedge P2 <> P1$

6: $\leftarrow Moderates(F0,U0) \wedge \neg Forum(F0)$

7: $\leftarrow Moderates(F0,U0) \wedge \neg User(U0)$

8: $\leftarrow Forum(F0) \wedge \neg MinModerator(F0)$

$MinModerator(F0) \leftarrow Moderates(F0,U0)$

9: $\leftarrow Moderates(F0,U0) \wedge Moderates(F0,U1) \wedge U1 <> U0$

10: $\leftarrow Participant(OID,F0,U0) \wedge \neg Forum(F0)$

11: $\leftarrow Participant(OID,F0,U0) \wedge \neg User(U0)$

12: $\leftarrow Forum(F0) \wedge \neg MinUserForum(F0)$

$MinUserForum(F0) \leftarrow Participant(P0,F0,U0)$

13: $\leftarrow Participant(OID1,F0,U0) \wedge Participant(OID2,F0,U0) \wedge OID1 <> OID2$

14: $\leftarrow Forum(F) \wedge Participant(P0,F,U0) \wedge Moderates(F,U0)$

15: $\leftarrow Participant(P0,F0,U0) \wedge HasInvited(P0,P0)$

16: $\leftarrow Forum(F) \wedge \neg AuxMain3(F)$

$AuxMain3(F) \leftarrow Forum(F) \wedge Participant(P0,F,U0) \wedge HasInvited(P0,P1) \wedge Participant(P1,F,U1)$

5.1. Violadores potenciales y reparadores

Como vimos anteriormente, una regla consiste en un conjunto de literales positivos, de literales negativos y un conjunto de *built-in literals*. Los literales positivos son los que violan la restricción que está conformando esa regla, mientras que los literales negativos reparan la restricción en caso de violación. En otras palabras, los literales negativos evitan la violación en caso de que se den todos los literales positivos de la restricción.

Decimos que un literal $p(\bar{X})$ es un violador potencial de una restricción ic si éste aparece en el cuerpo de la misma y no está negado. Denotamos como $V(ic)$ el conjunto de violadores potenciales de la restricción ic.

Por su parte, dada una restricción ic, hay un reparador $R_i(ic)$ por cada literal negativo $\neg L_i$ que se encuentre en el cuerpo de ic. Si el literal no hace referencia a una regla derivada, el reparador es, efectivamente, ese literal en su forma positiva, es decir, $R_i(ic) = \{L_i\}$. Si no, el reparador es el conjunto de literales del cuerpo de la regla derivada (todos ellos al mismo tiempo), es decir, $R_i(ic) = \{p_1(\bar{X}_1),\ldots,p_n(\bar{X}_n)\}$, donde cada $p_j(\bar{X}_j)$ es un literal positivo que aparece en el cuerpo de la regla derivada referenciada por L_i.

Vamos a ver un par de ejemplos. Tomemos, en primer lugar, la restricción 6 del modelo de la página anterior. *Moderates(F0,U0)* es el único violador potencial de esta restricción, mientras que *Forum(F0)* es su único reparador. Es decir, $V(ic6) = \{Moderates(F0,U0)\}$ y $R(ic6) = \{Forum(F0)\}$.

Tomemos ahora la restricción 16. Ésta vuelve a tener un solo violador potencial y un solo reparador. Su violador potencial es *Forum(F)*, pero su reparador es ahora un conjunto de literales: *{Forum(F) , Participant(P0,F,U0), HasInvited(P0,P1), Participant(P1,F,U1)}*. Es decir, $V(ic16) = \{ Forum(F)\}$ y $R(ic16) = \{ Forum(F) , Participant(P0,F,U0), HasInvited(P0,P1), Participant(P1,F,U1)\}$.

Si consideramos que existen una serie de hechos para este modelo, por ejemplo, {*Moderates(0,1) ,Forum(0)*}. Podemos decir que el hecho *Moderates(0,1)* viola la restricción 6, ya que es una instanciación de su violador potencial. Lo que pasa es que el también existe una instanciación para el reparador de la restricción 6, *Forum(0)*. Por lo tanto, así como *Moderates(0,1)* viola la restricción 6, *Forum(0)* la repara, y así se evita la violación.

¿Qué ocurre con la restricción 16? El reparador de 6, *Forum(0)*, la viola. Así que, para evitar la violación de 16, tendría que existir una instanciación de *Participant(P0,F,U0), HasInvited(P0,P1)* y *Participant(P1,F,U1)*. Habría que repararla, es decir, añadir al conjunto de hechos una instanciación (correcta) para el reparador de 16.

No hace falta que tengamos instanciados todos los reparadores de una restricción dada que está siendo violada para considerarla reparada; basta con disponer de uno de ellos.

Si tenemos una restricción que no tiene reparador, como es el caso de 13, al darse los hechos que instancian los violadores potenciales, no hay posibilidad de reparar esta restricción, a no ser que eliminemos los violadores del conjunto de hechos.

Pero estamos adelantando los acontecimientos. De momento, lo que vamos a necesitar es simplemente considerar los violadores potenciales y los reparadores (sin instanciación alguna) de cada restricción de nuestro modelo.

5.2. Nodos y arcos del multigrafo dirigido

Ahora que tenemos una lista de los violadores potenciales y de los reparadores de cada restricción, ya tenemos todo lo necesario para construir el grafo de dependencias. Para ser más concretos, dicho grafo es un multigrafo dirigido, ya que en lugar de aristas tendrá arcos (aristas con sentido) y un nodo podrá tener varios arcos de salida.

Cada nodo o vértice del grafo se corresponde con una restricción ic_i del modelo. Habrá un arco desde ic_i hasta ic_j etiquetado como $Rk(ici)$ si existe un predicado p tal que $p(X)$ pertenezca al conjunto de reparadores de ic_i y $p(\bar{X})$ pertenezca al conjunto de violadores potenciales de ic_j, es decir, tal que $p(\bar{X}) \in R_k(ic_i)$ y $p(\bar{Y}) \in V(ic_j)$.

Es decir, los arcos nos dicen qué nodo depende de cuál, en el sentido de que nos están diciendo los nodos que tal vez se vayan a violar al reparar otro. El grafo, pues, es en realidad un multigrafo porque un nodo puede tener varios reparadores.

A continuación mostramos el grafo de dependencias correspondiente al modelo que se encuentra en las dos primeras páginas de este capítulo.

Cada uno de los arcos está etiquetado con los predicados correspondientes a los literales del reparador de su predecesor que puede provocar la violación de su sucesor. En la figura 7, para

mayor claridad, los arcos aparecen etiquetados con letras y con los números de las restricciones (origen, final) entre paréntesis.

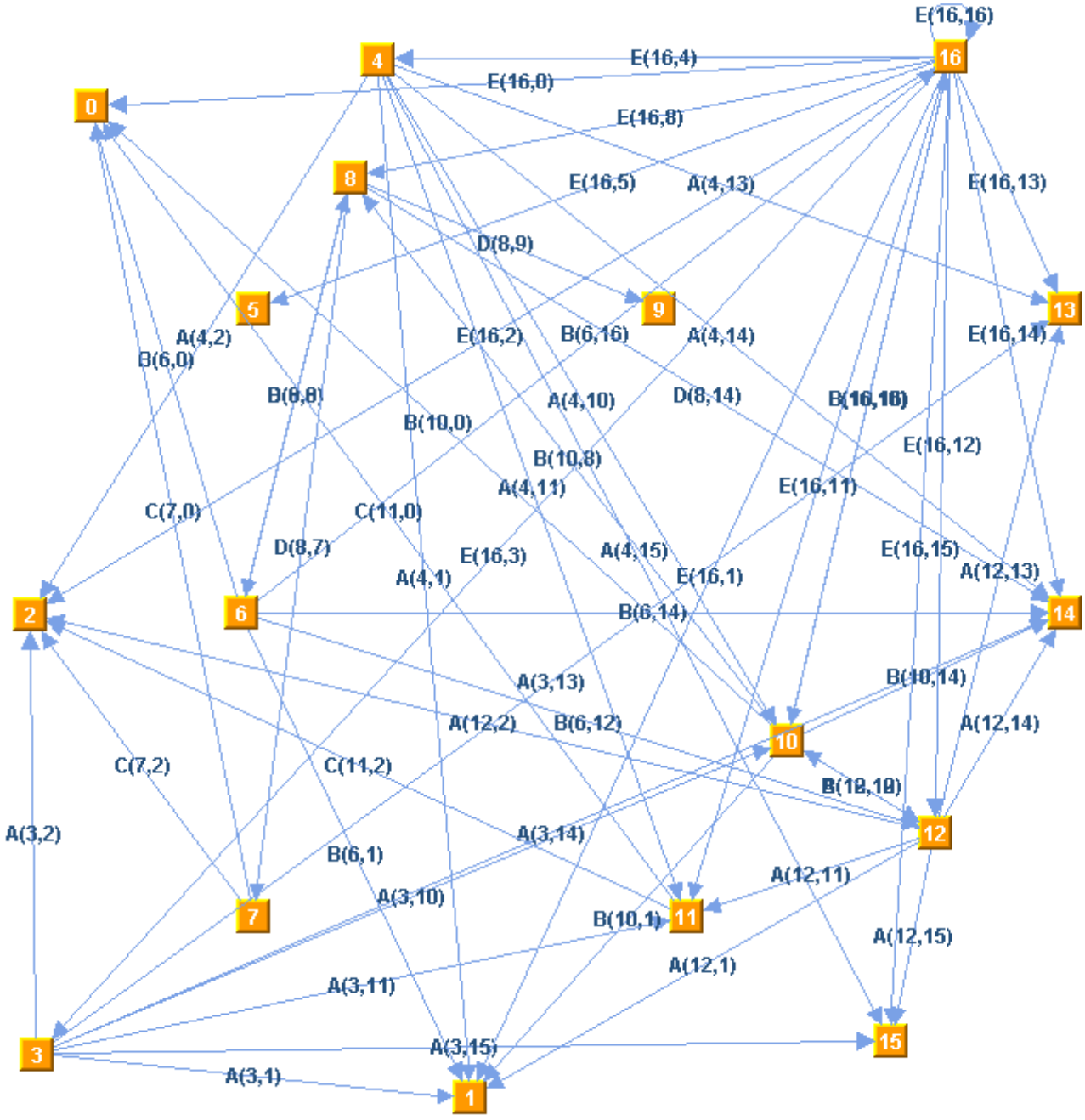


Figura 7: Multigrafo de dependencias dirigido

Las letras se corresponden con los predicados de los reparadores siguientes:

A: Participant
B: Forum
C: User
D: Moderates
E: Forum, Participant, HasInvited

En este grafo se puede ver, por ejemplo, que la reparación de la restricción 3 podría violar las restricciones 2, 13, 14, 10, 11, 15 y 1 (en orden horario respecto a su posición en el dibujo del grafo), porque el predicado *Participant* se encuentra al mismo tiempo en algún reparador de 3 (el único, de hecho) y en algún violador potencial de 2, 13, 14, 10, 11, 15 y 1.

5.3. Arcos superfluos

Hemos dicho que los arcos en el grafo representan la posible violación de la restricción sucesora al reparar la restricción predecesora, pero eso no es del todo cierto. Hay casos especiales en que un arco significa exactamente lo contrario: que la reparación de la restricción del nodo predecesor garantiza la no-violación de la restricción del nodo sucesor. Estos arcos reciben el nombre de arcos superfluos, porque, ya que anulan dependencias, podremos ignorarlos.

En la figura 7 del grafo anterior podemos encontrar dos pares de arcos superfluos. Son los correspondientes a los arcos entre los nodos de las restricciones 6 y 8 y entre los nodos de las restricciones 10 y 12.

Analicemos el primer par de arcos, los que hay entre 6 y 8.

6: $<-\ Moderates(F0,U0) \wedge \neg Forum(F0)$ *8:* $<-\ Forum(F0) \wedge \neg MinModerator(F0)$

$MinModerator(F0) <-\ Moderates(F0,U0)$

Cuando 1 es violada a causa de la hipotética inserción de *Moderates(f,u)* en nuestro conjunto de hechos, la introducción del reparador *Forum(f)* repara 6 pero a su vez garantiza que 8 no se violará, porque ya está reparado por *Moderates(f,u)*. No es que al añadir reparemos 8 directamente: hay que tener en cuenta la concordancia entre las constantes. Algo muy parecido ocurre con los arcos entre 10 y 12.

Formalmente, un arco r_i desde la restricción ic_i hacia ic_j es superfluo cuando $V(ic_j) = r_i\,\theta$ y hay algún reparador $R_k(ic_j)$ tal que $R_k(ic_j) = V(ic_i)\,\theta$, donde θ es un unificador de los conjuntos $V(ic_i) \cup r_i$ y $V(ic_j) \cup R_k(ic_j)$ que asigna una constante diferente para cada variable diferente.

Esto garantiza que ic_j nunca será violada tras reparar ic_i, ya que aunque los hechos insertados por ic_i violan en potencia ic_j, la restricción siempre se cumple por que ya disponíamos de un reparador en el conjunto de hechos. Y si tenemos un reparador (no hace falta que los tengamos todos instanciados), se evita la violación. Así pues, estos arcos pueden obviarse, ya que indican el final de una secuencia de reparaciones.

El aspecto del grafo de dependencias asociado al ejemplo tras la eliminación de los arcos superfluos sería como sigue:

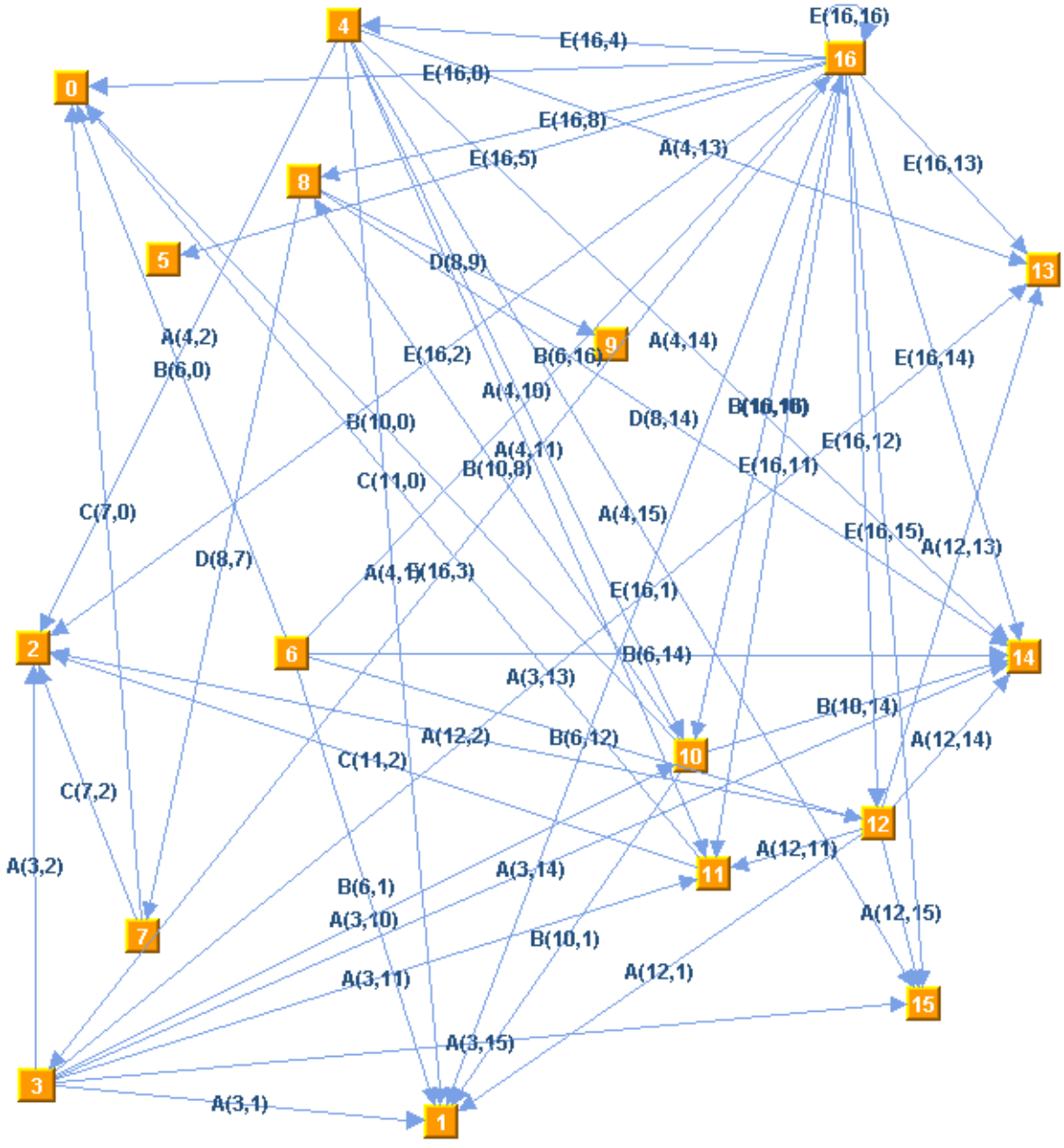


Figura 8: Multigrafo de dependencias dirigido sin arcos superfluos

6. DETECCIÓN Y ANÁLISIS DE CICLOS EN EL GRAFO

Hemos logrado construir un grado de dependencias que relaciona las restricciones de integridad de un modelo con los hechos que las violan potencialmente, con la forma que han de tener sus hechos reparadores y con las restricciones que hay que analizar tras realizar una reparación, ya que ésta ha introducido hechos que podrían violarlas.

Ya tenemos, pues, el marco de trabajo necesario para poder decir si un conjunto de literales concreto es satisfactible o no. La idea, como hemos dicho con anterioridad, es construir un ejemplo que no viole ninguna restricción de integridad y al mismo tiempo incluya una instanciación de los literales requeridos.

Sin embargo, acometer esta tarea en este punto puede representar un serio peligro, porque tal vez el proceso de construcción del ejemplo no acabe nunca. Es posible caer en un bucle infinito, generado por las dependencias del grafo que el proceso de validación utiliza como guía en su ejecución.

Digámoslo de una manera más formal. Sea $C = (p_1(ic_1,ic_2)\ldots,p_i(ic_i,ic_{i+1}),..,p_n(ic_n,ic_1)) = (r_1\ldots,r_i,..,r_n)$ una secuencia de arcos reparadores que define un ciclo en un grafo de dependencias G. La existencia de C implica que el reparador $r_i = p_i(ic_i,ic_j)$ de una restricción ic_i podría violar otras restricciones cuyos reparadores podrían violar ic_i de nuevo.

Para evitar este riesgo, el método Queralt define un protocolo para encontrar dichos bucles y analizar si son realmente infinitos o existe alguna secuencia de reparaciones que permita salir de él y continuar con el proceso de validación.

Dicho protocolo consiste en la aplicación de un conjunto de teoremas que determinan si un ciclo es finito o infinito. Por lo tanto, una vez identificados los ciclos en el grafo (hablaremos de ello en el capítulo 8.2.6.) podemos testear su condición finita/infinita a través de los tres teoremas que se exponen a continuación.

6.1. Teorema 1

Teorema 1. Un ciclo $C = (p_1(ic_1,ic_2)\ldots, p_i(ic_i,ic_{i+1}),..,p_n(ic_n,ic_1))= (r_1\ldots,r_i,..,r_n)$ es finito si:

$$\bigcup_{i=1}^{n}\left(\bigcup_{p(\bar{X})\in r_i} p\right) \subset \bigcup_{i=1}^{n}\left(\bigcup_{q(\bar{Y})\in V(ic_i)} q\right)$$

Intuitivamente, como la unión de reparadores de las restricciones en el ciclo es un subconjunto de la unión de violadores potenciales, al menos un violador potencial de una restricción ic_j en el ciclo es un predicado que no se actualiza en el mantenimiento del resto de las restricciones. Por lo tanto, como el conjunto de hechos que habíamos fabricado justo antes de iniciar el mantenimiento del ciclo era finito, ic_j sólo podrá ser violada un número finito de veces.

Lo veremos más claro con un ejemplo:

ic_1: $\leftarrow p(X) \wedge q(X) \wedge \neg r(X)$

ic_2: $\leftarrow r(X) \wedge \neg aux(X)$

$aux(X) \leftarrow p(X) \wedge Y<>X$

Vemos que existe un ciclo entre estas dos restricciones porque el reparador de la primera es un violador potencial de la segunda y viceversa: $C = (r(ic_1,ic_2), p(ic_2,ic_1))$. En este ciclo el reparador de la segunda restricción no vuelve a añadir el violador potencial $q(X)$. Por lo tanto, aunque la primera condición esté siendo violada por unos hipotéticos hechos $p(X) \wedge q(X)$ para una X dada, los reparadores de la segunda restricción podrían llevarnos a nuevas violaciones de la primera restricción sólo un número finito de veces (una por cada $q(a)$ presente en nuestro conjunto inicial de hechos, antes de iniciar la reparación del ciclo).

6.2. Teorema 2

Sin embargo, un ciclo puede ser finito aunque no cumpla la condición que se enuncia en el teorema 1. Pueden darse situaciones en las que todos los hechos creados durante el mantenimiento de un ciclo sean violadores potenciales, y aun así el ciclo puede ser finito.

Por ejemplo, en el modelo de Forums, Users y Participants encontramos un ciclo entre la restricción 10 y la restricción 16 que no satisface la condición del teorema 1 y es finito.

$$10: \quad \leftarrow Participant(OID,F0,U0) \wedge \neg Forum(F0)$$

$$16: \quad \leftarrow Forum(F) \wedge \neg AuxMain3(F)$$

$$AuxMain3(F) \leftarrow Forum(F) \wedge Participant(P0,F,U0) \wedge HasInvited(P0,P1) \wedge Participant(P1,F,U1)$$

Cuando añadimos un *Participant(a,b,c)*, la restricción 10 necesita que se inserte *Forum(b)* que, a su vez, viola la restricción 16. Para reparar esta violación se añadirían los hechos *Participant(a2,b,c2), HasInvited(a2,a3) y Participant(a3,b,c3)*, que no vuelven a violar la restricción 10 (porque *Forum(b)* lo está evitando).

A continuación enunciaremos un teorema que permitirá detectar este tipo de ciclos finitos, pero antes vamos a definir el concepto de variable libre: una variable X es libre en un reparador $R_i(ic)$ si $X \in variables(R_i(ic))$ pero $X \notin variables(V(ic))$.

Vemos que las variables libres en un reparador son el origen de los ciclos infinitos, ya que pueden perpetuar las violaciones al crear nuevas constantes diferentes de las que ya al principio violaban las restricciones del ciclo.

Teorema 2. Un ciclo $C = (r_1 \ldots, r_i, .., r_n)$, donde cada r_i corresponde a algún reparador $R_j(ic_i)$, es finito si se cumple que:

$$\forall i, 1 \leq i \leq n, \ \forall p \text{ tal que } p(X_1, \ldots, X_m) \in r_i \text{ y } p(Y_1, \ldots, Y_m) \in V(ic_{i+1}),$$
$$\forall k, 1 \leq k \leq m, \text{ siendo } X_k \text{ libre en } r_i \quad \rightarrow \quad Y_k \notin variables(r_{i+1})$$

La condición establecida por el teorema 2 garantiza que las variables libres en el reparador de la primera restricción no son propagadas por el reparador de la segunda. Como la condición se aplica para cada par de restricciones consecutivas, se garantiza que la reparación del ciclo no será eterna porque llegará un momento en que no se introducirán nuevas constantes.

Aplicando esta condición al ciclo entre las restricciones 10 y 16 podemos concluir que éste es finito, ya que las constantes añadidas por las variables libres del reparador de 16 no aparecen en el reparador de 10, lo que significa que no se propagan nuevas constantes en el ciclo.

6.3. Teorema 3

No obstante, aunque un ciclo no cumpla las condiciones definidas por el teorema 1 y el teorema 2 no puede ser considerado un ciclo infinito. Por ejemplo, el ciclo formado por la restricción 3, 10 y 16 no es infinito.

$3: \quad \leftarrow HasInvited(P0,P1) \wedge \neg IsParticipant(P0)$

$IsParticipant(P0) \leftarrow Participant(P0,F0,U0)$

$10: \quad \leftarrow Participant(OID,F0,U0) \wedge \neg Forum(F0)$

$16: \quad \leftarrow Forum(F) \wedge \neg AuxMain3(F)$

$AuxMain3(F) \leftarrow Forum(F) \wedge Participant(P0,F,U0) \wedge HasInvited(P0,P1) \wedge Participant(P1,F,U1)$

Podemos ver que las variables libres del reparador de 16 se propagan en el reparador de 3 pero no en 10 y, por lo tanto, no hay una nueva violación de 16. Necesitamos otra condición para saber si un ciclo es infinito.

Definición: Sea $C = (r_1 \ldots, r_i, .., r_n)$ un ciclo en G, donde cada r_i corresponde a algún reparador $R_j(ic_i)$. Sea $V(ic_i)=\{p_1(\bar{X}_1),\ldots,p_m(\bar{X}_m)\}$. Bajo estas premisas,

$$Facts(ic_1) = (V(ic_1) \cup r_1)\,\theta_1$$

$$Facts(ic_i) = \bigcup_{k=1}^{t} r_i\,\theta_k \cup Facts(ic_{i-1}),\ i > 1$$

donde θ_1 es una sustitución que asigna una constante distinta a cada variable, cada $\theta_k = \theta_j \cup \theta'_j$ es una de las t posibles sustituciones tal que $Facts(ic_{i-1}) \vDash (p_1(\bar{X}_1),\ldots,p_m(\bar{X}_m)\,\theta_j$ y θ'_j asigna una nueva constante a cada variable X tal que $X \in variables(r_i)$ y $X \notin variables(V(ic_i))$ si $\theta'_j \neq \emptyset$, si no, $\theta_k = \emptyset$.

Teorema 3. C es finito si, para cada posible restricción inicial ic_i,

$\exists k, 1 \leq k \leq n$, tal que $Facts(ic_k) = Facts(ic_{k-1})$

Intuitivamente vemos que para cada $i > 1$ el conjunto $Facts(ic_i)$ extiende el conjunto $Facts(ic_{i-1})$ al tener en cuenta los reparadores necesarios para satisfacer ic_i. Por lo tanto, la condición $Facts(ic_k) = Facts(ic_{k+1})$ garantiza que la restricción ic_{k+1} no se viola y el proceso de mantenimiento de la integridad de las restricciones en el ciclo no será eterno.

Los teoremas anteriores, pues, nos permiten determinar si el proceso de mantenimiento de un esquema conceptual puede no acabar o bien tal posibilidad queda descartada, al no hallar ciclos infinitos. Hay que tener en cuenta, no obstante, que este conjunto de teoremas no es completo debido a la indecidibilidad del problema.

6.4. El grafo de dependencias resultante

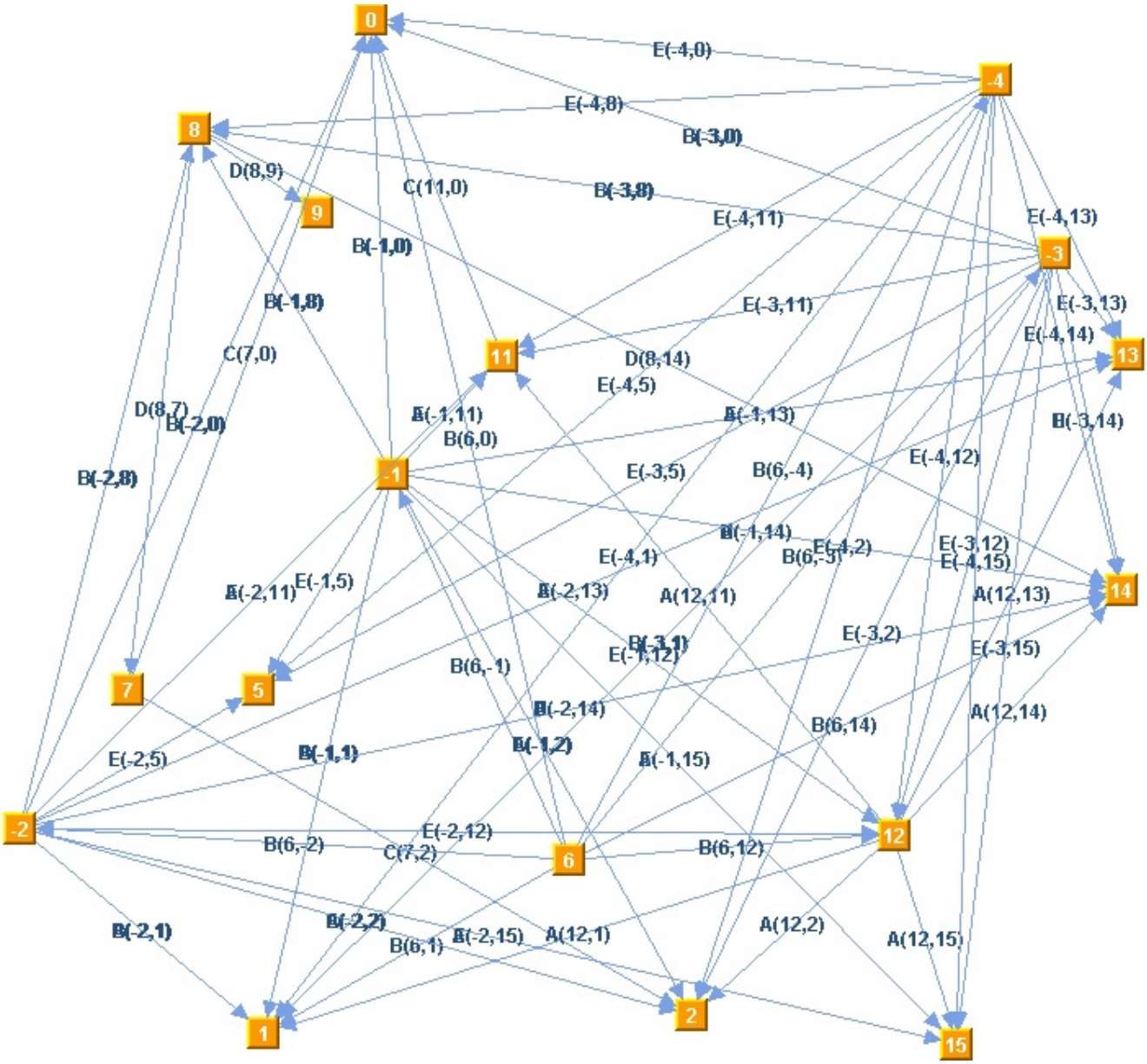
Debido a las necesidades del algoritmo de validación que estudiaremos en el siguiente capítulo, necesitamos una última versión del grafo, donde cada ciclo haya sido compactado y atomizado en un solo nodo. En cada nuevo nodo, que etiquetamos con un número negativo para facilitar su identificación, incidirán los arcos de entrada que incidían sobre los nodos del ciclo que representa. A su vez, de cada uno saldrán los arcos que salían de los nodos de su ciclo.

No obstante, seguimos manteniendo los ciclos aparentemente erradicados como subgrafos anidados en estos nuevos nodos. No perdemos información alguna, aunque no se muestre en el grafo por razones obvias de simplicidad.

Esta transformación es útil por la manera en que se repararán las violaciones de los ciclos. En resumen, si se viola alguna restricción de un ciclo éste ha de ser reparado completamente. De manera que se trata un ciclo como si fuera un solo nodo, a efectos de su reparación.

Además, también a instancias del algoritmo de validación, habremos de conocer cuál es el grado de entrada de cada nodo. Para el grafo resultante del modelo de ejemplo, éstos son sus grados de entrada:

Grado 0:	*Nodo 6*	*Grado 9:*	*Nodo 2*
Grado 1:	*Nodos 7, 9, -1, -2, -3, -4*	*Grado 10:*	*Nodo 0*
Grado 4:	*Nodo 5*	*Grado 11:*	*Nodo 15*
Grado 5:	*Nodo 12*	*Grado 15:*	*Nodo 1*
Grado 7:	*Nodos 8, 11, 13*	*Grado 16:*	*Nodo 14*



7. VALIDACIÓN DEL ESQUEMA

Una vez hemos logrado determinar que todos los ciclos del modelo o esquema conceptual son finitos (como ocurre en el modelo de ejemplo de Forum, User y Participant), podemos aprovechar la traducción a lógica realizada con anterioridad para definir un proceso de *reasoning* más eficiente que otros. Un proceso de *reasoning* se refiere al mecanismo concreto que se ejecuta para saber si un esquema conceptual es correcto, mediante la comprobación la satisfactibilidad o insatisfactibilidad de ciertas propiedades para dicho modelo.

Una manera bastante conocida de solucionar este problema es construir un ejemplo, es decir, crear un conjunto de hechos que satisfagan al mismo tiempo las propiedades que se desee satisfacer (*goal*) y todas las restricciones de integridad el propio modelo.

En este capítulo se expone el mecanismo de *reasoning* del método Queralt [3], que se apoya en la infraestructura creada en los pasos anteriores detallada en capítulos previos.

7.1. Satisfacción de las propiedades deseadas (goal)

La base de hechos que debemos construir demuestra que el esquema cumple ciertas propiedades deseadas (las que el analista desea comprobar). A menudo llamaremos *goal* a estas propiedades y utilizamos la letra G para referirnos a él. Se asume que G es un conjunto de literales positivos y negativos correspondientes a predicados del esquema y literales built-in.

Los literales positivos de G son (o representan, porque pueden contener variables) hechos que son necesarios para satisfacer G, mientras que sus literales negativos identifican hechos que la base de hechos no debe contener de ningún modo. Por su parte, los literales built-in definen condiciones sobre los valores que las variables de los literales de G pueden tomar.

El primer paso del mecanismo de *reasoning* consiste en determinar, simplemente, qué hechos son necesarios para satisfacer G, sin tener en cuenta si éstos están violando alguna restricción de integridad del esquema.

La clave de este primer paso es escoger correctamente las constantes que puede tomar cada variable. Cada alternativa define una nueva posibilidad de satisfacer G, es decir, una base de hechos alternativa.

Para asignar constantes a variables utilizaremos un sistema definido en [4] llamado *Variable Instantiation Patterns* (VIPs). Estos VIPs garantizan que el número de bases de hechos alternativas a ser consideradas es finito, al tener en cuenta sólo aquellas instanciaciones de variables que son relevantes para el esquema. Es decir, los VIPs garantizan que si no se encuentra una solución mediante la instanciación de las variables en G usando únicamente las constantes que proporcionan, entonces la solución no existe.

En este PFC utilizaremos dos tipos de VIPs, en función de las propiedades sintácticas de G y del propio esquema:

- Negation VIP: para esquemas sin comparaciones (es el caso del modelo del cap. 5)
- Order VIP: para esquemas con comparaciones (es el caso del modelo del cap. 4)

En el modelo de ejemplo de Forum, User y Participant no hay comparaciones, por lo tanto, se utiliza el Negation VIP. Con este VIP, a la hora de instanciar un determinado literal, cada variable debe ser instanciada tanto con cada una de las constantes que se han usado previamente como con una nueva. Si estamos en un punto en el que hemos usado n constantes, obtendremos $n+1$ instanciaciones diferentes del literal: $n+1$ hechos para añadir a $n+1$ nuevas bases de hechos. Por ejemplo, si queremos instanciar $P(X)$ y hasta ahora hemos utilizado las constantes 0 y 1, deberemos crear los hechos $P(0)$, $P(1)$ y $P(2)$.

Formalmente, sea $G \leftarrow p_1(\bar{X}_1) \wedge \ldots \wedge p_n(\bar{X}_n) \wedge \neg q_1(\bar{Y}_1) \wedge \ldots \wedge \neg q_m(\bar{Y}_m) \wedge b_1 \wedge \ldots \wedge b_s$, donde p_i y q_j son predicados no derivados y b_k son literales built-in. Sea θ una posible sustitución obtenida a partir de la instanciación de $variables(G)$ tal que $\forall i, 1 \leq i \leq s$ se cumplen las condiciones establecidas por $b_i\,\theta$. Bajo estas premisas,

- El conjunto o base de hechos necesarios para satisfacer G es BH = $\{p_1\theta,\ldots,p_n\theta\}$
- El conjunto de hechos no deseados para satisfacer G es HND = $\{q_1\theta,\ldots,q_m\theta\}$

Por ejemplo, asumamos que el analista desea comprobar si la asociación *Moderates* puede tener instancias en el esquema conceptual del ejemplo del capítulo 5. Esto equivale a comprobar la

satisfactibilidad de la propiedad $G \leftarrow Moderates(F,U)$, por que se intentará construir alguna base de hechos que satisfaga esta propiedad y no viole ninguna restricción de integridad. Por el momento, en este primer paso en el que nos limitamos a satisfacer el *goal*, creamos dos bases de hechos: $BH_1=Moderates(0,0)$; $BH_2=Moderates(0,1)$.

7.2. El mantenimiento de la integridad del modelo

Una vez hemos añadido hechos que satisfacen el *goal*, hemos de velar por que no haya ninguna restricción de integridad que esté siendo violada. Si no hubiera ninguna violación para la base de hechos que estuviéramos considerando, habríamos acabado.

Normalmente, no obstante, hemos de realizar un proceso en el que debemos añadir, con cierto criterio, nuevos hechos que eviten violaciones, hasta conseguir que no haya violación alguna. A veces, esto será imposible, por lo que habrá que descartar la base de hechos que estemos considerando en ese momento y escoger otra alternativa, en caso de haberla. Acabaremos tan pronto como encontremos una base de hechos que no viole ninguna restricción ni contenga ningún hecho indeseable o cuando se nos hayan acabado las alternativas, en cuyo caso determinaremos que la propiedad del *goal* es insatisfactible.

Ésta es la idea que subyace tras el mecanismo de *reasoning* del método Queralt, y el que está implementado en la herramienta Aurus, basada en el método CQC [4].

El método CQC adolece de ciertas limitaciones que provocan pérdida de eficiencia en casos prácticos. Para soslayar estas limitaciones, el método Queralt sugiere utilizar las reglas lógicas obtenidas en el proceso de traducción y el grafo de dependencias. Ya hemos visto que una primera mejora consistía en la detección de ciclos infinitos. Una segunda mejora consiste simplemente escoger la restricción a mantener en cada momento de una forma más astuta, estableciendo un orden.

En primer lugar, tendrán prioridad los nodos con menor grado de entrada, es decir, con menos arcos incidentes. Además, se podrán escoger únicamente aquellos nodos que representen restricciones de integridad que pueden ser potencialmente violadas, es decir, cuyos predecesores hayan sido mantenidos. Esto se traduce en que tendremos en todo momento un

registro con los nodos que se pueden visitar y los que no, así como de su grado de entrada. De esta manera garantizamos que una restricción no será considerada (con la pérdida de tiempo que ello supondría) hasta que todas sus predecesoras hayan sido reparadas.

Una restricción de integridad ic debe ser reparada si sus violadores potenciales se hallan en la base de hechos. El mantenimiento de ic implica la inclusión de sus reparadores en la base de hechos en construcción.

Es importante tener en cuenta que ic puede ser violada por diferentes instanciaciones de sus violadores potenciales, todas ellas presentes en la base de hechos. Eso significa que hay que reparar todas ellas, añadiendo diferentes hechos nuevos al ejemplo en construcción. Estos hechos nuevos probablemente contendrán constantes nuevas, creadas de acuerdo al VIP correspondiente según la sintaxis del *modelo* y el *goal*.

Si una restricción es violada y no tiene reparadores, la base de hechos en construcción queda descartada porque es imposible evitar tal violación. Sólo podemos considerar una base de hechos alternativa sin los hechos que violan dicha restricción.

Esto se traduce en aplicar un algoritmo de *backtracking* que crea un árbol virtual donde cada una de las hojas es una posible base de hechos, tal vez errónea. Deberemos explorar ese espacio de búsqueda hasta hallar una hoja válida o haber agotado las opciones. La forma en que realicemos la poda (descartar ramas del árbol) será crucial para la eficiencia del algoritmo. En cada *paso atrás* que el algoritmo de *backtracking* se vea obligado a realizar, habrá que considerar reparadores alternativos.

Formalmente, sea $ic \leftarrow p_1(\bar{X}_1) \wedge \ldots \wedge p_n(\bar{X}_n) \wedge \neg q_1(\bar{Y}_1) \wedge \ldots \wedge \neg q_m(\bar{Y}_m) \wedge b_1 \wedge \ldots \wedge b_s$ la restricción seleccionada del grafo de dependencias para mantener, donde p_i y q_j son predicados no derivados y b_k son literales built-in.

Sea BH_i el conjunto de hechos necesario en un momento dado. Sea $EvalV(ic)$ y $EvalR(ic)$ el conjunto de literales built-in que aparecen en el cuerpo de ic y en el cuerpo de la regla a partir de la que $R_i(ic)$ ha sido obtenido, respectivamente.

Bajo estas premisas, BH_{i+1} se calcula como sigue:

si $R_i(ic) =$ y $BH_i \vDash (p_1(\bar{\bar{X}}_1) \wedge \ldots \wedge p_n(\bar{X}_n) \wedge \neg q_1(\bar{Y}_1) \wedge \ldots \wedge \neg q_m(\bar{Y}_m) \wedge b_1 \wedge \ldots \wedge b_s)\ \theta_j$

entonces *backtrack* (*ic* no se puede reparar y hay que buscar alternativas)

si no $BH_{i+1} = \cup_{k=1}^{t} R_i(ic)\theta_k \cup BH_i$

si $\exists q_i \in HND$ tal que $q_i \in BH_{i+1}$

entonces backtrack (*ic* no se puede reparar y hay que buscar alternativas)

Hay que tener en cuenta que cada $\theta_k = \theta_j \cup \theta'_j$ es una sustitución tal que $BH_i \vDash (V(ic) \wedge EvalV(ic))\ \theta_j$ y θ'_j es una de las posibles sustituciones obtenidas a partir de una instanciación de todas las variables presentes en $variables(R_i(ic)) \setminus variables(V(ic))$, tal que la evaluación $EvalR(R_i(ic))\ \theta'_j$ da cierto, si $\theta'_j \neq \emptyset$, si no, $\theta_k = \emptyset$.

7.2.1. Ejemplo de validación de una *query*

Veamos cómo se realizaría la validación de la *query* que hemos considerado en el punto anterior: Moderates(F,U), que al satisfacerla (primer paso de la validación) nos proporciona dos bases de hechos. BH_1=*Moderates(0,0);* BH_2=*Moderates(0,1).*

En principio, tomaríamos *Moderates(0,0)* para empezar pero, como sabemos que va a ser insatisfactible, vamos a ejecutar *Moderates(0,1)*, que permitirá ilustrar mejor el funcionamiento del algoritmo.

A continuación se expone la ejecución del algoritmo con ciertas licencias en favor de la claridad del ejemplo.

Tomamos la segunda BH

BH_2 = *Moderates(0,1)*

¿Se viola alguna restricción? Vamos a verlo. Empezamos por considerar los nodos con menor grado de entrada en el grafo. El primero es 6.

¿La ic6 se viola? Sí. La reparamos y añadimos Forum(0)

BH_3 = *Moderates(0,1) ^ Forum(0)*

Ahora podemos tomar 7, 9, -1, -2, -3 o -4. Seleccionamos la primera: 7.

¿La ic7 se viola?Sí. Añadimos User(1).

BH_4 = *Moderates(0,1) ^ Forum(0) ^User(1)*

Seguimos con la restricción 9. No se viola porque no hay dos instancias de Moderates con instancias de User diferentes.

Seguimos con el nodo -1, que representa el ciclo: 10-16-3.

¿Qué hacemos ahora? Hay que mantener y reparar el ciclo completamente antes de continuar.

Miramos si 16 se viola. Es el caso. Añadimos Participant(2,0,1) ^ HasInvited(2,3) ^ Participant(3,0,1)

Sabemos que estos hechos violan la restricción 13, pero el algoritmo no lo podrá saber hasta llegar a 13, para lo cual falta visitar unos cuantos nodos.

Salimos del bucle porque ya está reparado (lo comprobamos analizando 3, 10 y de nuevo 16)

BH_5 = *Moderates(0,1) ^ Forum(0) ^User(1)^ Participant(2,0,1) ^ HasInvited(2,3) ^ Participant(3,0,1).*

Pasamos por -2, -3, -4, 5, 12, 8 y 11 y no encontramos nuevas violaciones hasta 13, en que se detecta una violación sin posibilidad de reparación y se hace backtrack.

Volvemos a 16 para escoger otro reparador y la base de hechos queda como:

BH_6 = *Moderates(0,1) ^ Forum(0) ^User(1) ^ Participant(2,0,1) ^ HasInvited(2,3) ^ Participant(3,0,4).*

Salimos del bucle.

Pasamos por -2, -3, -4, 12, 8 y se detecta una violación en 11 que se repara insertando User(4).

BH_7 = *Moderates(0,1) ^ Forum(0) ^User(1) ^ Participant(2,0,1) ^ HasInvited(2,3) ^ Participant(3,0,4) ^ User(4)*

El algoritmo seguiría trabajando hasta darse cuenta de que los hechos actuales violan la restricción 14. Cuando llegue este momento, volvería a hacer backtracking hasta probar con otro reparador de 16.Finalmente, la base de hechos que conseguiría satisfacer todas las condicione es como sigue:

BH_8 = *Moderates(0,1) ^ Forum(0) ^User(1) ^ Participant(2,0,3) ^ HasInvited(2,4) ^ Participant(4,0,5) ^User(3)^User(5)*

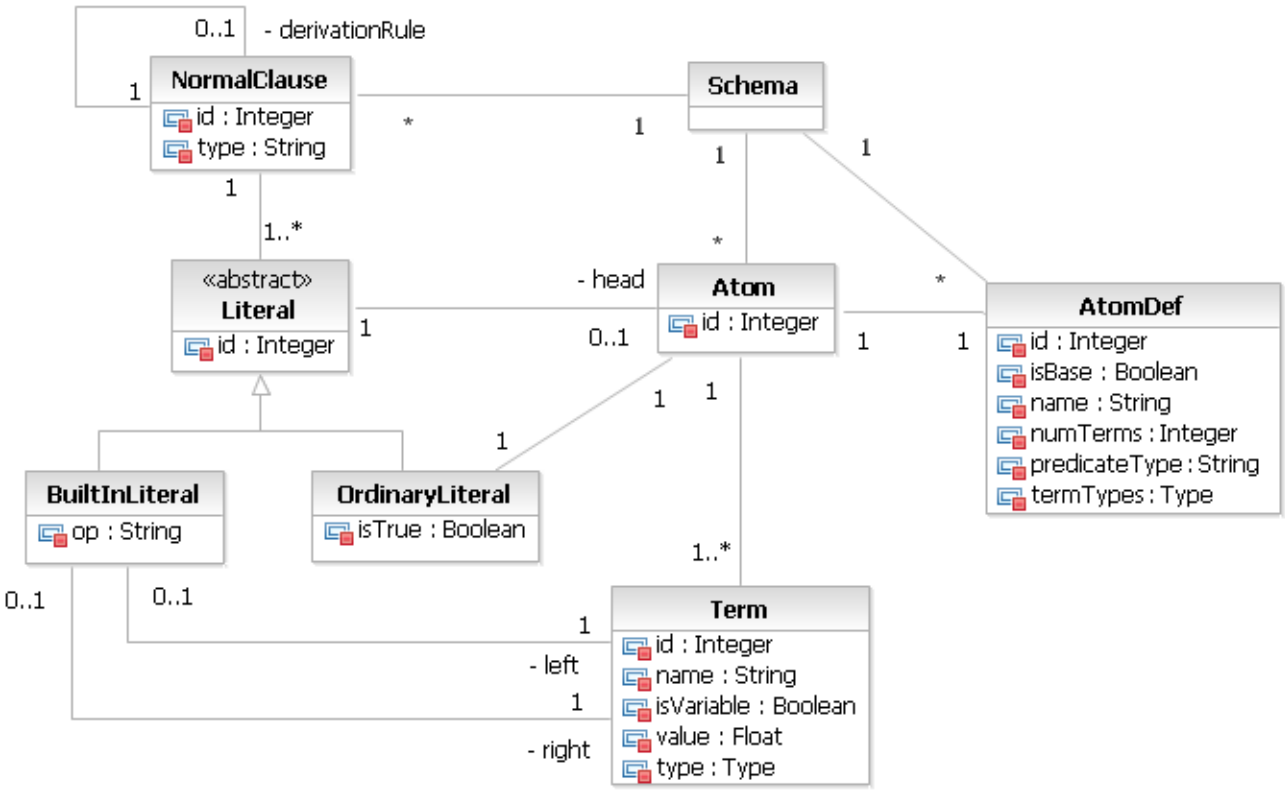
8. LOS ENTRESIJOS DE LA HERRAMIENTA AURUS

En este capítulo se presenta la especificación de la herramienta Aurus, así como comentarios sobre el diseño y la especificación de la misma, con el objetivo de dar una visión más profunda de sus características. Analizaremos la arquitectura de la herramienta, sus funcionalidades y la manera de interactuar con la misma.

8.1. Especificación de la herramienta

8.1.1. El metamodelo lógico

Éste es el fragmento del diagrama de clases correspondiente a la lógica. Se corresponde con la explicación del metamodelo lógico del capítulo 2.3.



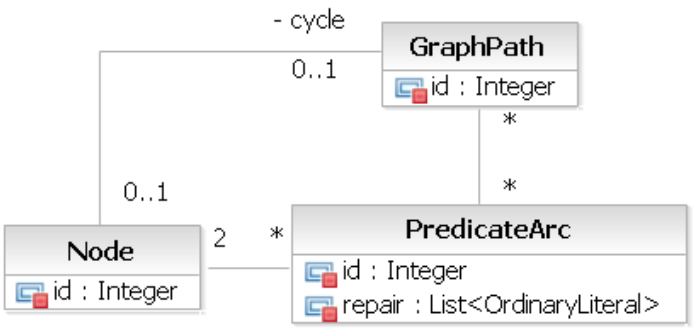
La clase *AtomDef* implementa el concepto "predicado". Como se puede observar, tiene un campo llamado *predicateType*. Éste permite distinguir entre predicados que representan una clase, predicados que representan una asociación, etc. Además, un *AtomDef* sabe cuántos términos pueden tener sus átomos, así como el tipo de cada uno de ellos. Si el predicado hace referencia a un head, *isBase* será falso.

Otro detalle importante es que si un término tiene valor (*value!=null*), entonces es una constante y, por lo tanto, *isVariable* es falso.

Además, dado que Aurus sólo trabajará con modelos en los que las reglas derivadas de las reglas no tengan, a su vez, reglas derivadas, es útil mantener una asociación recursiva sobre *NormalClause* para representar esta relación.

8.1.2. El grafo de dependencias

El grafo de dependencias tiene dos elementos fundamentales: nodos (o vértices) y arcos. En nuestro caso, los nodos representan restricciones de integridad.



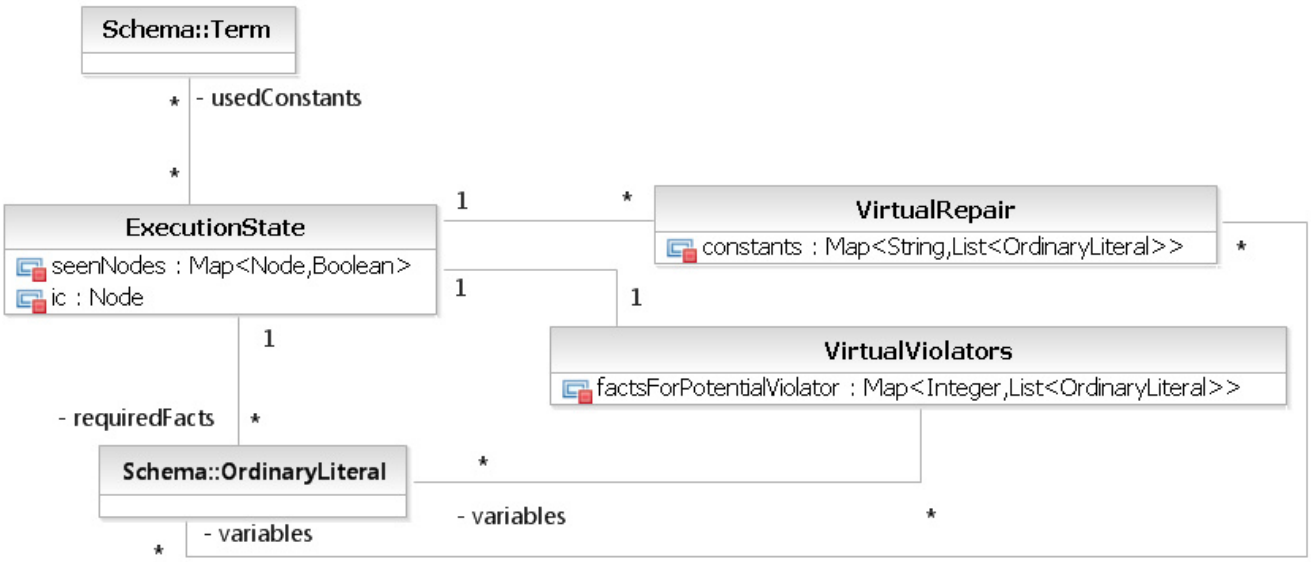
Una vez tenemos los nodos creados y añadidos al grafo original, el siguiente paso natural sería fabricar los arcos. Es necesario realizar un análisis previo, porque los arcos se crean en función de los violadores potenciales y los reparadores de cada nodo, tal como se explica en el capítulo 5.2. A partir de este análisis se crean los arcos, que son de tipo *PredicateArc* y guardan el reparador concreto que provoca la dependencia entre nodos.

¿Por qué se utiliza una clase Nodo y no se instancia, directamente, la clase NormalClause (siempre que ésta representara una regla no derivada)? La razón es que un nodo expresa más información. Concretamente, necesitamos distinguir entre nodos normales y nodos que encapsulan un ciclo. Como se ha introducido en el capítulo 6.4., hay nodos que contienen un subgrafo (el ciclo).

Los subgrafos en cuestión serán instancias de la clase *GraphPath*, es decir, caminos del grafo formados por arcos, es decir, una sucesión de *PredicateArcs*. En este caso, además, los caminos serán ciclos.

8.1.3. Validación del esquema

El último fragmento de diagrama que vamos a considerar hace referencia al proceso de validación.



Durante el proceso de validación necesitaremos almacenar cierta información porque tal vez debamos recuperarla al hacer un paso atrás en el *backtracking*. A cada paso del algoritmo tendremos un estado de ejecución (*ExecutionState*).

¿Qué se debe guardar en un estado de ejecución? Entre otras cosas, los violadores virtuales y los reparadores virtuales para un nodo dado, y los nodos que ya hemos visitado.

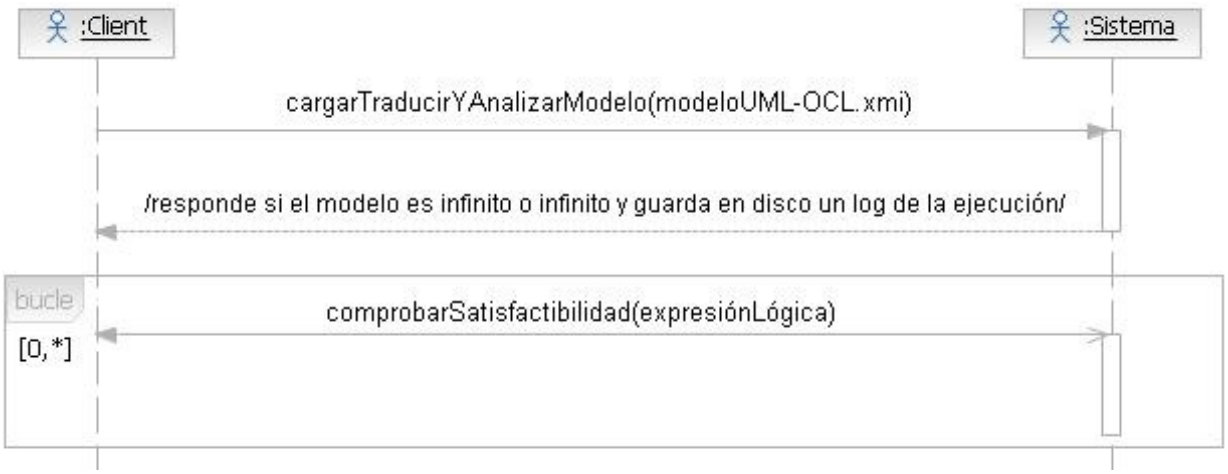
Si recordamos la explicación del capítulo 7.2, necesitamos ordenar los nodos a visitar y saber cuáles hemos visitado para recorrer el grafo de forma eficiente. Por eso existe el atributo *seenNodes.*

Los violadores virtuales y los reparadores virtuales representan el conjunto de combinaciones de literales que violan o reparan una restricción (nodo) *ic* de un estado de la ejecución. Cada *ExecutionState* tiene un conjunto de violadores virtuales (*VirtualViolators*). A su vez, cada *ExecutionState* tiene un conjunto de conjuntos de violadores virtuales (*VirtualRepair*), uno por cada alternativa formal de reparación, es decir, uno por cada reparador genérico diferente de la *ic* en cuestión.

Además, cada ExecutionState debe ser consciente de la base de hechos que está construyendo (requiredFacts) y las constantes que existen en ella (usedConstants).

8.1.4. Casos de uso

Se podría resumir la manera de utilizar Aurus y sus funcionalidades en un solo párrafo: el usuario introduce un modelo en UML/OCL, la herramienta genera su traducción a lógica, crea el grafo de dependencias, lo analiza y, en caso de no existir ciclos infinitos, el usuario puede introducir *queries*, obteniendo la respuesta correspondiente.



Tal como se aprecia en el diagrama de secuencia anterior, esta herramienta tiene únicamente un caso de uso con un curso principal de los acontecimientos y apenas un flujo alternativo (finalizar antes de validar ninguna *query*).

Actor: Analista	**Sistema: Aurus**
1. El usuario introduce un modelo especificado en UML/OCL.	
	2. Aurus responde si el modelo es finito.
3. El usuario introduce una query.	
	4. Aurus responde si la query es satisfactible y muestra un ejemplo en caso afirmativo.
5. El usuario vuelve al punto 3 o acaba.	
	6. Aurus guarda en disco un log con los resultados de la validación y termina su ejecución.

Además, durante la ejecución de Aurus se crean y se almacenan en disco un conjunto de documentos de texto. Se trata de 4 logs o dietarios que registran la actividad de la ejecución.

El paso 2 del caso de uso es el que genera los archivos xxx_translation_log.txt, xxx_graphConstruction_log.txt y xxx_graphAnalysis_log.txt, que registran la traducción, la construcción del grafo (nodos, aristas y leyenda para los dibujos de los grafos) y análisis del grafo (arcos superfluos y enumeración y análisis de los ciclos). Adicionalmente, en el paso 2 se muestra un dibujo de los grafos de dependencia.

El paso 6 del caso de uso es el que genera xxx_validationResults.txt, que contiene, para cada query formulada por el analista, si es satisfactible o no y, en caso de serlo, un ejemplo que lo demuestra.

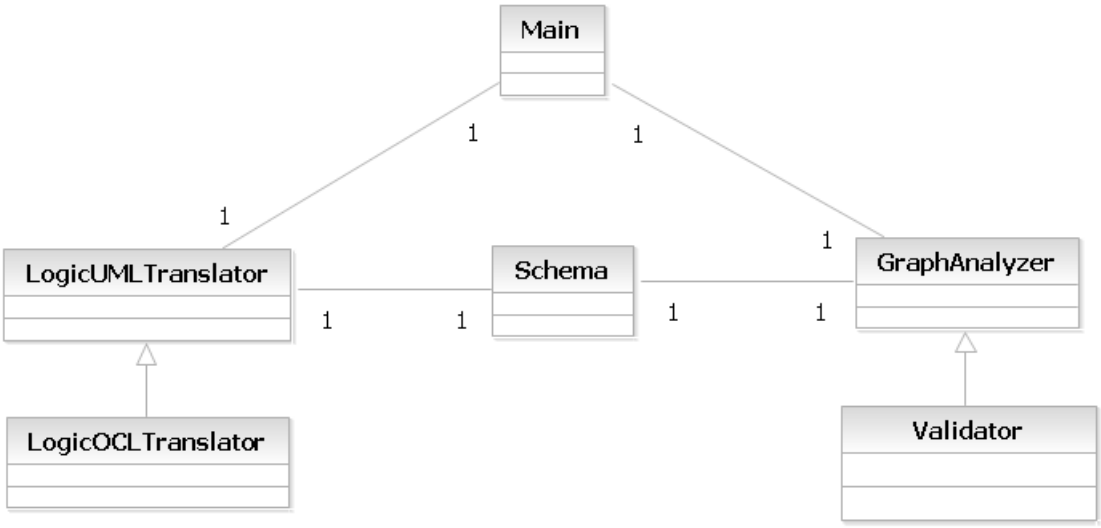
8.2. Diseño e implementación

En esta sección se comentan algunas peculiaridades de la implementación, así como sucesos destacables durante la realización de la misma.

Como se ha comentado con anterioridad, el código de Aurus está escrito íntegramente en Java y se desarrolló en el entorno Eclipse. Además, se han utilizado la librería que implementa el núcleo de EinaGMC, la librería del conversor XMIConverter, la API de UML2 y OCL2, la librería JGraphT para crear y manipular grafos y la librería JGraph para poder visualizarlos.

8.2.1. Visión general

En primer lugar, presentamos una visión general de los componentes (que han acabado siendo módulos en el diseño, paquetes con las clases Java correspondientes en su interior).



Tal como se aprecia en el diagrama, desde el *Main* se puede acceder a los componentes que hacen las veces de traductor (*LogicUMLTranslator* y *LogicOCLTranslator*). Éstos crean una instancia del metamodelo lógico (*Schema*) que aparece en el centro, es decir, crean el modelo lógico correspondiente. Una vez completado el modelo, la clase *GraphAnalyzer* se encarga de construir y analizar el grafo de dependencias asociado al modelo lógico. Por último, la clase

Validator, que hereda atributos y operaciones de *GraphAnalyzer*, es la encargada de realizar el test de satisfactibilidad para cada *query*.

Aunque en el diagrama no aparece gráficamente, todas estas clases son en realidad *singleton*. Sólo pueden tener una instancia, ya que sólo tienen que preocuparse de trabajar con un modelo simultáneamente.

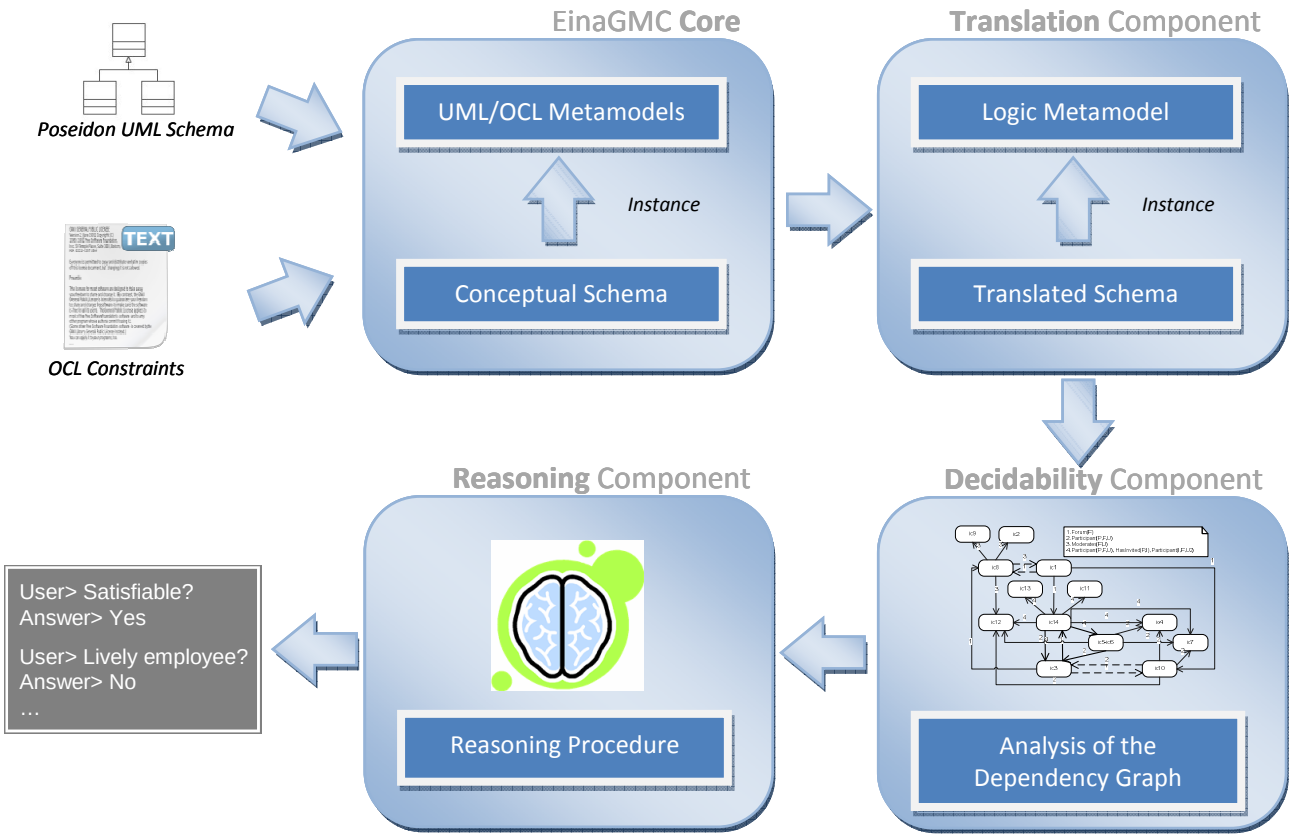
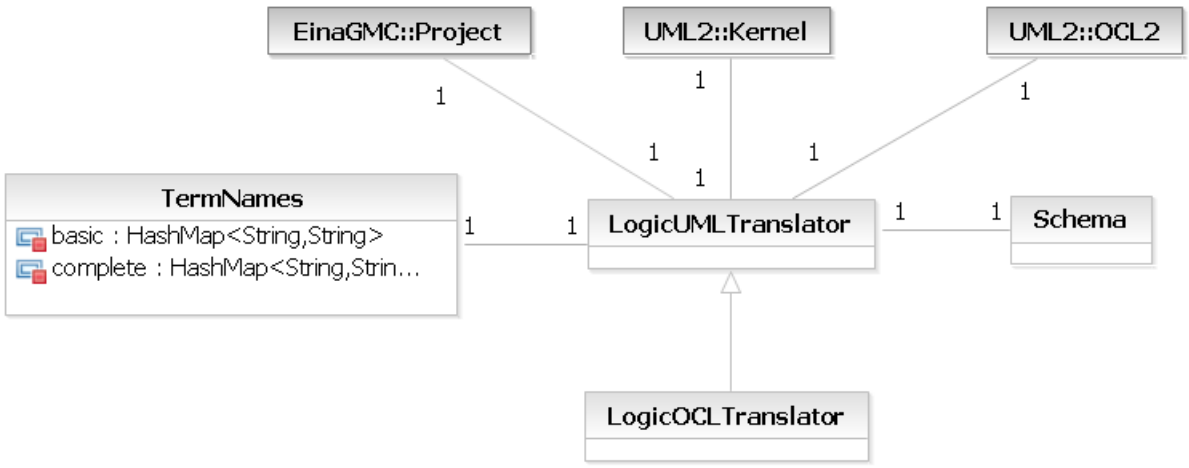


Figura 9: Esquema general de la arquitectura y funcionamiento de Aurus

8.2.2. Traducción a lógica

La versión en código Java de las reglas de traducción propuestas en el capítulo 4 se puede encontrar en la clase UMLTranslator y OCLTranslator, dentro del paquete Translator, tal como representa este diagrama.



Podemos observar que el dúo *LogicUMLTranslator-LogicOCLTranslator* crea el esquema lógico a partir de lo que la librería EinaGMC llama Project. El Project contiene las interfaces para acceder a los elementos UML y OCL del modelo. Los traductores obtienen los elementos que necesitan para efectuar las reglas de traducción pertinentes y, si es necesario, acceden a sus atributos u operaciones.

Estos elementos a los que nos referimos son objetos de tipo *UmlClass*, *Property*, *Type*, *Association*, *AssociationClass*, *Constraint*, etc, propios de los metamodelos de UML y OCL. Tales metamodelos se encuentran ya implementados y su especificación, que no incluiré en esta memoria, se puede encontrar en [5] y [6], respectivamente. Únicamente he añadido una breve referencia en forma de caja negra, como en el caso de la clase Project. Para aquellos que estén más familiarizados con lenguajes orientados a objetos, esta referencia se traduce en un *import* en el código de la clase.

Aparece también en el diagrama la clase *TermNames*. Esta clase existe para dar apoyo a la implementación: concretamente mapea nombres de términos lógicos con nombres de predicados. Del mismo modo que ocurre en el resto de clases que se encuentran en los diagramas de estas páginas, he omitido algunos atributos utilizados por cuestiones técnicas concretas que no son relevantes en esta especificación.

8.2.2.1. Traducción de UML a lógica

Por lo que respecta a la implementación de la traducción UML-lógica, fue muy importante entender correctamente el metamodelo de UML, en el que había que bucear para obtener la información necesaria para traducir a lógica. Podría destacar, por ejemplo la manera en que están representadas las jerarquías, que no es demasiado intuitiva. Se trata de un metamodelo muy completo y con muchos recovecos.

Por lo que respecta a las reglas en sí, las reglas de traducción asociadas a las cardinalidades fueron, sin duda, las más difíciles de implementar. Generar los *built-in-literals*, es decir, los literales de comparación, no es trivial. Hay que mantener en todo momento un mapeo entre los términos que hay que igualar y los que hay que desigualar.

Por otra parte, creo que es importante comentar la decisión que se tomó de modificar ligeramente la sintaxis de las reglas derivadas. Si consideramos algunas reglas que tienen reglas derivadas en la traducción del modelo de los departamentos, originalmente se transcribía de la siguiente manera, donde el nombre de las variables no importa realmente, donde hay X puede haber Z, mientras se respete la concordancia a nivel de cláusula normal:

3: *<- HasInvited(P0,P1) ^ ¬IsParticipant(P0)*
IsParticipant(P) <- Participant(P,F,U)
4: *<- HasInvited(P0,P1) ^ ¬IsParticipant(P1)*
8: *<- Forum(F) ^ ¬MinModerator(F)*
MinModerator(X) <- Moderates(X,U)

Sin embargo, ha acabado siendo así:

3: *<- HasInvited(P0,P1) ^ ¬IsParticipant(P0)*
IsParticipant(P0) <- Participant(P0,F0,U0)
4: *<- HasInvited(P0,P1) ^ ¬IsParticipant(P1)*
IsParticipant(P1) <- Participant(P1,F0,U0)
8: *<- Forum(F0) ^ ¬MinModerator(F0)*
MinModerator(F0) <- Moderates(F0,U0)

¿Cuáles son las diferencias? Semánticamente, por supuesto, no hay ninguna. Pero fijémonos en las reglas derivadas de las restricciones 6 y 7. En realidad, es la misma regla derivada, tal como se refleja en la transcripción original, pero replicada con otras variables. Se ha forzado que las variables concuerden, cosa que, en principio, es totalmente innecesaria. Se trata de facilitar el análisis posterior, así como la propia construcción de las reglas, imponiendo concordancia a nivel de reglas y sus reglas derivadas al mismo tiempo.

8.2.2.2. Traducción de OCL a lógica

La implementación de la traducción de OCL a lógica me resultó mucho más difícil, tal como estaba previsto, y prácticamente me ocupó 2 meses. Así como el código de la clase UMLTranslator tiene unas 1000 líneas de código, la clase que implementa OCLTranslator tiene unas 2000. Es un dato que, más allá de resultar anecdótico, da una idea bastante precisa de la diferencia entre UML y OCL, respecto a la dificultad en la implementación de la traducción.

No es sólo que el metamodelo de OCL sea bastante más enrevesado que el de UML, por lo que bucear en él representa una tarea peliaguda, sino que el propio lenguaje es muy expresivo (aun estando simplificado), de manera que existen múltiples posibilidades para expresar invariantes y todas ellas deben estar previstas.

Por ejemplo, algo que no está especificado en el artículo de Anna Queralt y Ernest Teniente *Reasoning on UML Class Diagrams with OCL Constraints* [2] es qué ocurre cuando una operación de selección tiene dos variables de iteración en lugar de una sola. Por ejemplo, el invariante:

context Avaria inv 12:
--Todos los técnicos externos que intervienen en una misma avería son de una sola emrpesa
self.intervencio.tecnic -> select(t1,t2 | t1<>t2 and t1.oclIsTypeOf(Extern) and t2.oclIsTypeOf(Extern) and t1.oclAsType(Extern).empresa<>t2.oclAsType(Extern).empresa) -> size() = 0

Se traduce como:

<- Avaria(AV,E0,DA0) ^ Intervencio(IN0,AV,DA1,TE0) ^ Tecnic(TE0) ^ Intervencio(in1,AV,DA2,TE1) ^ Tecnic(TE1) ^ Extern(TE0) ^ TreballaPer(TE0,EM0) ^

Empresa(EM0) ^ Extern(TE1) ^ TreballaPer(TE1,EM1) ^ Empresa(EM1) ^ TE0<>TE1 ^ EM0<>EM1

En el ejemplo anterior aparecen, además, las operaciones *oclIsTypeOf* y *oclAsType*. ¿Cómo traducir estas expresiones? Tampoco está definido en el artículo. Se tuvo que pensar la manera de hacerlo y la conclusión fue hacerlo añadiendo un literal tomando el predicado correspondiente a la subclase con la misma variable que el literal que hace referencia a la superclase. Lo mismo se puede aplicar para las dos operaciones *oclIsTypeOf* y *oclAsType*.

Tampoco está definido qué pasa cuando queremos traducir una expresión booleana compleja. Traducir $value_1 > value_2$ es muy fácil: $Tr\text{-}Path(value_1) > Tr\text{-}Path(value_2)$. ¿Pero qué ocurre cuando tenemos el siguiente invariante?

```
context Equip inv 10:
  --Un equipo no puede tener dos averías que se solapen temporalmente
  self.avaria -> select(a1,a2 | a1<>a2 and not (a1.oclIsTypeOf(Reparada) and
  a2.oclIsTypeOf(Reparada) and not
   (a1.oclAsType(Reparada).dataFi>=a2.inici.data and
  a2.oclAsType(Reparada).dataFi>=a1.inici.data)) and not
  (a1.oclIsTypeOf(Reparada) and not a2.oclIsTypeOf(Reparada) and
  a2.inici.data>a1.oclAsType(Reparada).dataFi) and not
  (a2.oclIsTypeOf(Reparada) and not a1.oclIsTypeOf(Reparada) and
  a1.inici.data>a2.oclAsType(Reparada).dataFi)) ->  size() = 0
```

Se traduce como:

<- Equip(E) ^ Avaria(AV0,E,DA0) ^ Avaria(AV1,E,DA1) ^ AV0<>AV1 ^
¬AuxBooleanExp10_0(AV0,AV1) ^ ¬AuxBooleanExp10_2(AV0,AV1) ^
¬AuxBooleanExp10_4(AV0,AV1)

AuxBooleanExp10_1(AV0,AV1) <- Avaria(AV0,E,DA0) ^ Reparada(AV0) ^
ReparadaDataFi(AV0,DATA1) ^ Avaria(AV1,E,DA1) ^
Reparada(AV1) ^ ReparadaDataFi(AV1,DATA2) ^ Data(DA1) ^
DataData(DA1,DAT1) ^ Data(DA0) ^ DataData(DA0,DAT2) ^
DATA1>=DAT1 ^ DATA2>=DAT2

AuxBooleanExp10_0(AV0,AV1) <- Avaria(AV0,E,DA0) ^ Reparada(AV0) ^ Avaria(AV1,E,DA1) ^
Reparada(AV1) ^ ¬AuxBooleanExp10_1(AV0,AV1)
AuxBooleanExp10_3(AV0,AV1) <- Avaria(AV1,E,DA1) ^ Reparada(AV1)
AuxBooleanExp10_2(AV0,AV1) <- Avaria(AV1,E,DA1) ^ Data(DA1) ^ DataData(DA1,DAT3) ^
Avaria(AV0,E,DA0) ^ Reparada(AV0) ^
ReparadaDataFi(AV0,DATA3) ^
¬AuxBooleanExp10_3(AV0,AV1) ^ DAT3>DATA3
AuxBooleanExp10_5(AV0,AV1) <- Avaria(AV0,E,DA0) ^ Reparada(AV0)

AuxBooleanExp10_4(AV0,AV1) <- Avaria(AV0,E,DA0) ^ Data(DA0) ^ DataData(DA0,DAT4) ^
Avaria(AV1,E,DA1) ^ Reparada(AV1) ^
ReparadaDataFi(AV1,DATA4) ^
¬AuxBooleanExp10_5(AV0,AV1) ^ DAT4>DATA4

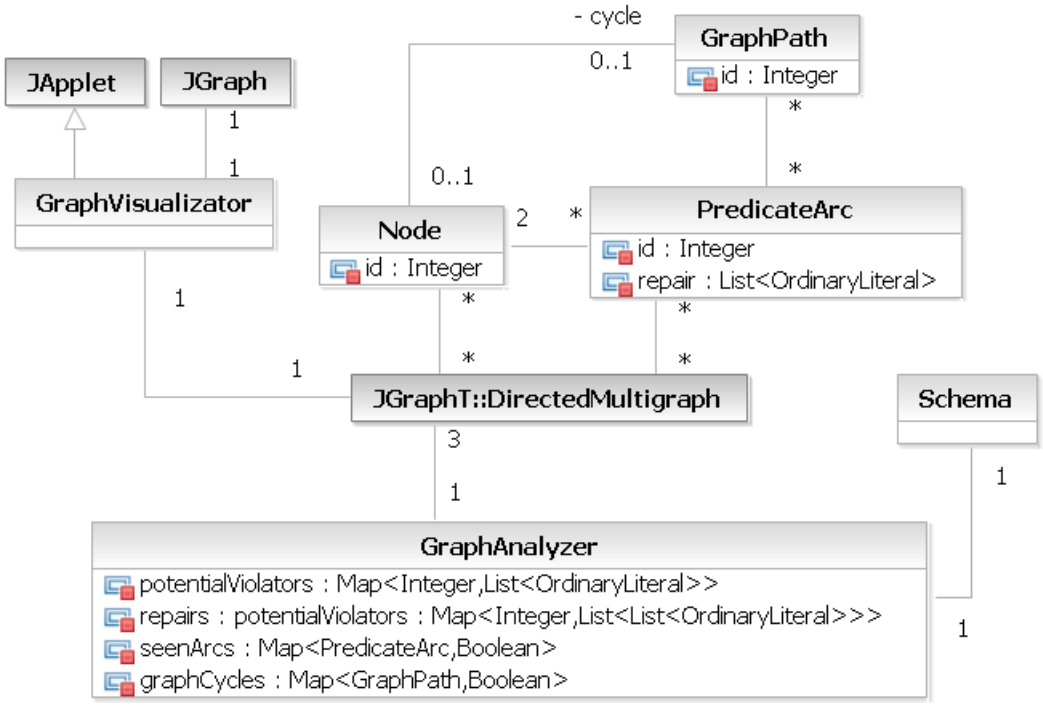
Sin embargo, aunque se ha llevado a cabo su implementación, Aurus no acepta este tipo de restricciones. Concretamente, la herramienta sólo puede validar modelos cuya traducción a lógica carezca de reglas con más de una regla derivada. Es una limitación del método Queralt que provoca que no todos los modelos conceptuales puedan ser validados con este sistema.

8.2.3. Generación y análisis del grafo de dependencias

La primera tarea de la clase *GraphAnalyzer* no es otra que la de crear los nodos del grafo: uno por cada regla o cláusula normal no derivada del esquema lógico. Una vez creados, se añaden al grafo, que es una instancia de la clase *DirectedMultigraph* de la librería *JGraphT*.

En el diagrama de la página siguiente, se puede observar que la asociación entre *GraphAnalyzer* y *DirectedMultigraph* tiene cardinalidad 3 en la parte del grafo. Esto es porque en realidad tendremos 3 grafos: el grafo original sin simplificar, el grafo sin arcos superfluos y el grafo con los ciclos compactados (éste es el que usaremos para la validación).

El diseño de las clases que participan en el proceso de creación y análisis del grafo de dependencias de muestra a continuación.



Una vez tenemos los nodos creados y añadidos al grafo original, el siguiente paso natural sería fabricar los arcos (*PredicateArc*), proceso para el que se necesitan los violadores potenciales y los reparadores de cada nodo, tal como se explica en el capítulo 5.2. Ésta es la razón de ser de los dos primeros atributos de *GraphAnalyzer*: son mapeos de identificadores de nodo con sus violadores potenciales o reparadores.

Una vez el grafo original está construido, se analiza (el atributo *seenArcs* se utiliza en este punto). Como resultado de la eliminación de los arcos superfluos tendremos el segundo grafo. Y como resultado de la identificación de los ciclos, crearemos nuevos nodos y los asociaremos con el ciclo en cuestión, de tipo *GraphPath*. Más tarde el analizador decidirá cuáles de estos ciclos son finitos o infinitos y los guardará en un mapa llamado *graphCycles*.

8.2.3.1. Generación del grafo

En general, podría decir que la generación del grafo fue la parte más fácil con respecto a la implementación, excepto tal vez las clases que hacen referencia a la visualización del grafo utilizando la librería JGraph [14].

Se estuvo investigando cuál era la mejor opción para implementar un multígrafo dirigido etiquetado, que era lo que se necesitaba, y se escogió JGraphT [13] porque, realmente, no había muchas alternativas para este tipo de grafo compatibles con Java. Se pensó, además, que ahorraría tiempo de implementación, en caso de haber deseado hacer la implementación del grafo "a medida", y al mismo tiempo sería más eficiente. Además, presentaba la ventaja de poder visualizar el grafo e incluso manipular su aspecto mediante un applet y mediante la librería JGraph.

No obstante, la versión gratuita de JGraph no incluye funcionalidades de *layout*, lo que se traduce en que, al visualizar el grafo, todos sus nodos se concentraban en un solo punto. Parece poco relevante, pero en el momento que el modelo requiere un grafo con 30 nodos, la pérdida de tiempo es considerable. Decidí, entonces, implementar una suerte de "dispersador de nodos", dentro de la clase *GraphVisualizator* que proporciona una presentación del grafo algo más comprensible.

8.2.3.2. Detección y análisis de los ciclos en el grafo

Esta parte de la implementación se basa en seis algoritmos: para detectar ciclos, para distinguir ciclos equivalentes, para encapsularlos en un nodo y uno para aplicar cada uno de los teoremas que comprueban si un ciclo es infinito.

Detectar ciclos en un grafo puede ser una tarea complicada, sobretodo cuando el grafo es grande. Es importante no perder eficiencia. Por este motivo estuve investigando si existía alguna implementación óptima que realizara esta tarea. Efectivamente, existen para grafos "normales". Por desgracia, el nuestro es un multigrafo dirigido y no encontré ningún algoritmo adecuado.

No tuve más remedio que diseñar un algoritmo hecho a medida, cuya función es encontrar caminos que empiecen y acaben en el mismo nodo. Tras realizar varias pruebas, puedo concluir que el resultado ha sido satisfactorio y el tiempo de respuesta es bastante bueno.

El inconveniente es que el algoritmo que detecta ciclos no distingue entre ciclos equivalentes. Es decir, consideraba que el ciclo A-B-C-A y el ciclo B-C-A-B eran diferentes, ya que el nodo inicial y final es diferente. Por este motivo necesité una función para "limpiar" los resultados e identificar qué ciclos eran equivalentes, proporcionando una única versión de cada ciclo.

Una vez hecho esto, un nuevo algoritmo se encarga de encapsularlos en un nodo. Primero se copia el grafo original. Para cada ciclo se construye un nuevo nodo en el que colocamos el ciclo, sobre la copia; se toman del grafo original los arcos de entrada de cada nodo del ciclo y se hacen incidir en el nuevo nodo; y hacemos lo mismo para los arcos de salida. Ya tenemos todos los ciclos encapsulados en un nuevo grafo.

A continuación, se puede analizar cada uno de los ciclos para saber si son finitos o infinitos. Para ello se aplican tres algoritmos, uno para cada uno de los teoremas.

Claramente el algoritmo que más trabajo me costó fue el tercero. Además de ser el más difícil de entender, es el más complicado de implementar. Básicamente se trata de simular su mantenimiento varias veces, una por cada nodo del ciclo, imaginando que la restricción que representa es violada. La simulación que se realiza incluye considerar siempre el peor caso, que consiste en instanciar los hechos reparadores siempre con constantes nuevas. Si aun considerando el peor caso no iteramos más de una vez sobre el ciclo, podemos concluir que, efectivamente, el ciclo es finito.

Para entender mejor qué es lo que hace este teorema veamos un ejemplo. Consideremos el ciclo formado por las restricciones 3, 10 y 16 del ejemplo de *Forum, User, Participant*.

3: $\leftarrow HasInvited(P0,P1) \wedge \neg IsParticipant(P0)$

$IsParticipant(P0) \leftarrow Participant(P0,F0,U0)$

10: $\leftarrow Participant(OID,F0,U0) \wedge \neg Forum(F0)$

16: $\leftarrow Forum(F) \wedge \neg AuxMain3(F)$

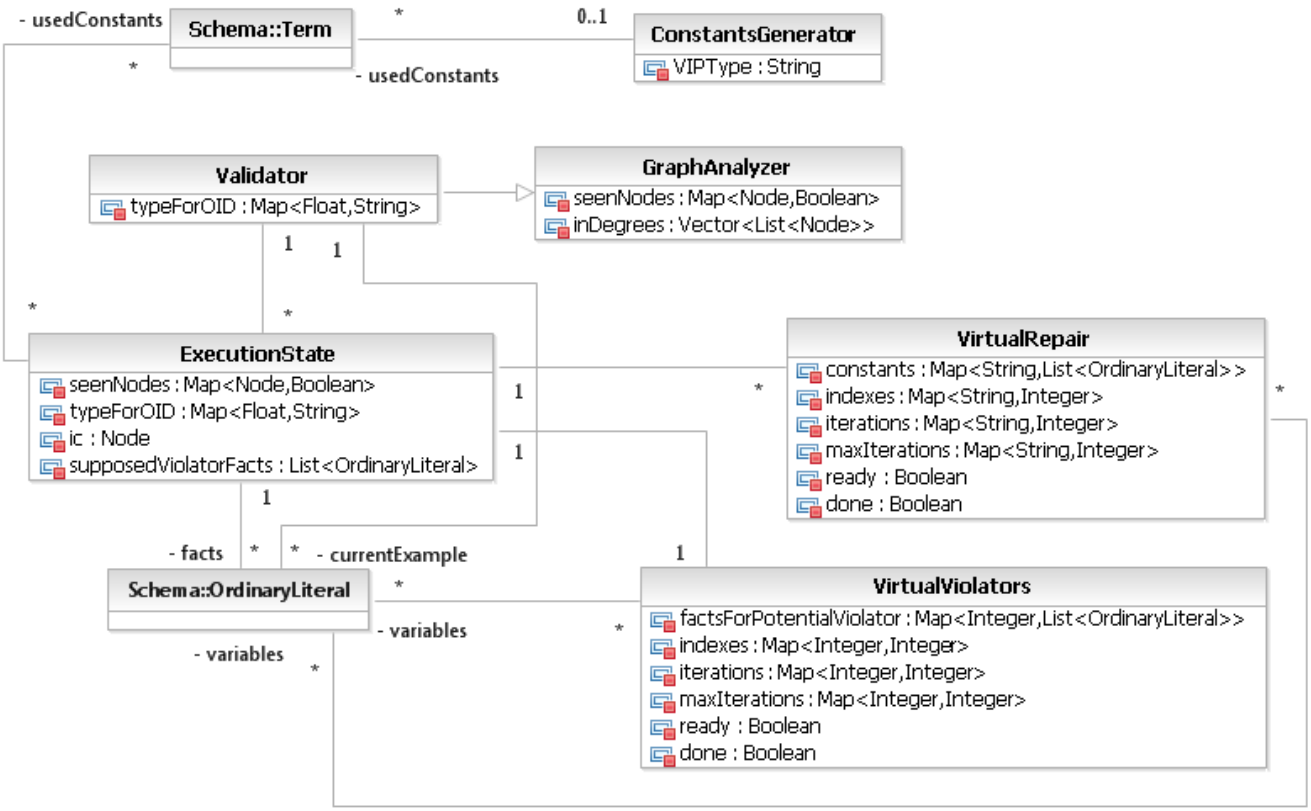
$AuxMain3(F) \leftarrow Forum(F) \wedge Participant(P0,F,U0) \wedge HasInvited(P0,P1) \wedge Participant(P1,F,U1)$

Primero, el algoritmo introduce los hechos HasInvited(1,2) para violar ic 3.
Comprueba que 3 se viola y la repara añadiendo Participant(1,3,4).
Comprueba que 10 se viola y la repara añadiendo Forum(3).
Comprueba que 16 se viola y la repara añadiendo Participant(5,3,6),
HasInvited(5,7), Participant(7,3,8).

Ahora volvemos a mirar la restricción 3 y, como vemos que no se viola, ya podemos simular la siguiente violación, empezando por10.
Cuando hayamos acabado de simular el mantenimiento del ciclo para 10 y 16 y hayamos acabado antes de completar una segunda iteración, podemos concluir que el ciclo es finito.

8.2.4. Validación del esquema

La clase *Validator*, que hereda atributos y funciones de *GraphAnalyzer*, es la encargada de implementar el algoritmo de validación. Se trata de una clase *singleton* porque habrá una sola instancia por modelo a validar. Eso sí, puede tener N estados de ejecución concurrentes (ExecutionState), aunque obviamente sólo uno estará activo en cada momento y los demás estarán convenientemente almacenados.



El atributo *inDegrees* registra el grado de entrada de cada nodo y nos permite seleccionar en cada momento la restricción adecuada.

8.2.4.1. La ventaja de almacenar estados de ejecución

En realidad, la gran ventaja de *ExecutionState* es que optimiza el mecanismo de poda en el *backtracking* del algoritmo de mantenimiento.

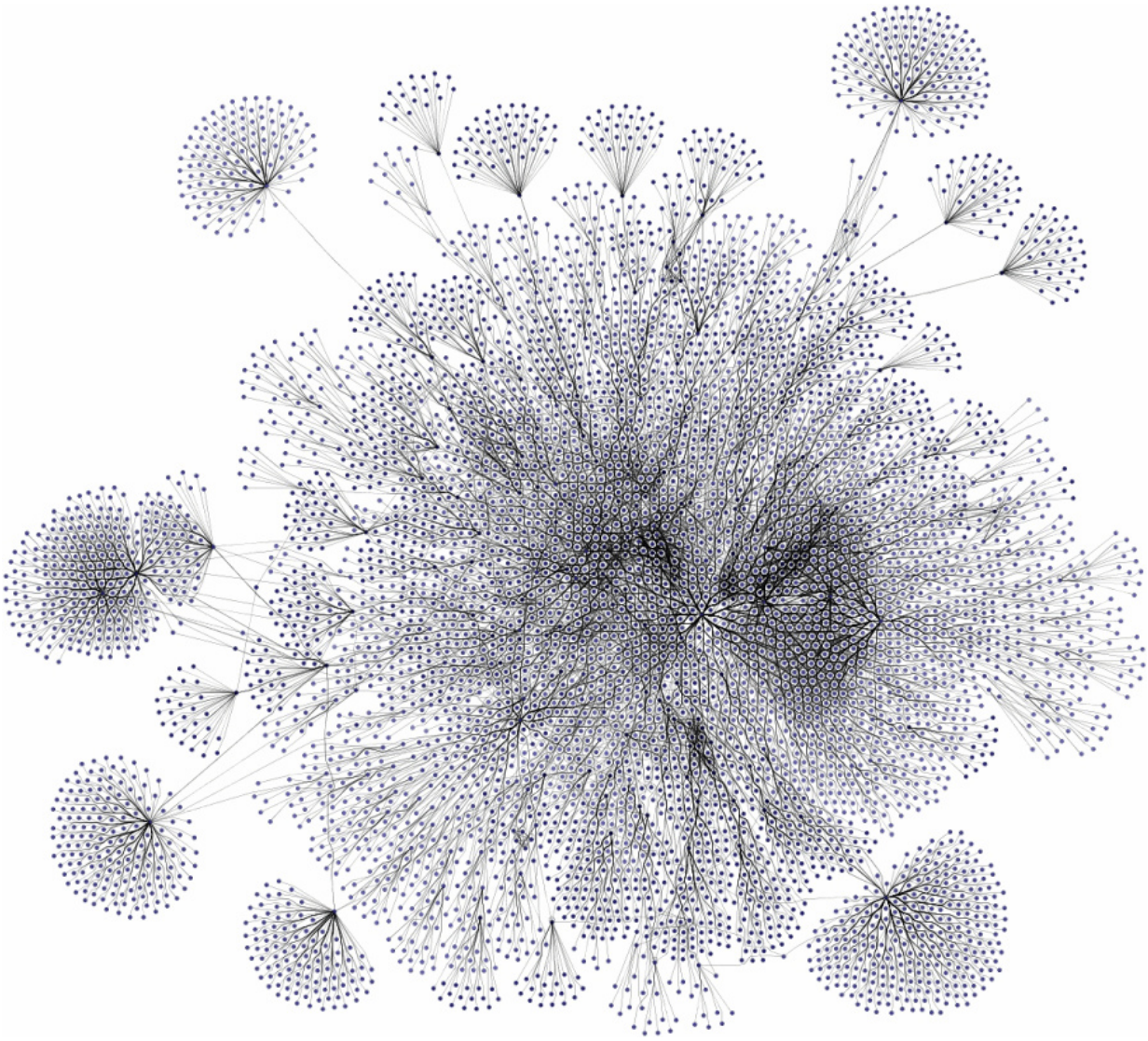
Cuando en un algoritmo basado en *backtracking* identificamos que una rama del árbol de búsqueda no nos va a llevar a ninguna solución válida, podamos esa rama. Hay muchas maneras de podar. La solución fácil (y tonta) es podar la mínima parte posible y seguir explorando la rama vecina. Pero si resulta que la rama defectuosa tiene su origen varias bifurcaciones atrás, estaremos perdiendo el tiempo inútilmente. Una solución más inteligente (y complicada) pasa por tener algún mecanismo que nos lleve al origen del problema, podar toda esa gran rama que sólo nos está estorbando, y seguir. Esa es la gran ventaja de *ExecutionState*, porque permite salvar un estado de la ejecución (sería como un punto de libro) y restaurar otro estado, según convenga.

Ya se ha hablado de la clase *ExecutionState* durante la explicación de la especificación, aunque se omitieron algunos detalles de implementación. Por ejemplo, el atributo *seenNodes* sirve para controlar qué nodos se han visitado y cuáles faltan por visitar. Cuando hayamos visitado todos los nodos y tengamos una base de hechos válida, podemos responder que la *query* era válida.

8.2.4.2. El problema de la explosión combinatoria

Los violadores virtuales y los reparadores virtuales también suponen una gran ventaja para la implementación. La idea es fabricar los posibles reparadores para un nodo a partir de todos los hechos violadores de ese nodo y una vez los tenemos, los vamos probando. Una manera de hacerlo sería: “consideremos todos los hechos violadores del nodo; para cada combinación de hechos violadores del nodo, consideremos todos sus posibles reparadores; intentemos aplicar un reparador; si no funciona, probemos el siguiente; etc”.

Si tenemos en cuenta que por cada variable libre que encontramos en un reparador pueden existir muchas constantes posibles para construir un hecho reparador, si tenemos muchos reparadores formales posibles con varios literales cada uno con unas cuantas variables cada literal, tenemos esto:



Tememos una explosión combinatoria que más nos valdrá evitar para que el procesador de la máquina donde se ejecuta Aurus no se sature. Por eso utilizamos índices y otras estructuras de control que conforman los violadores y reparadores virtuales: para poder generar las posibilidades una a una, bajo demanda del algoritmo de mantenimiento, y no perder el hilo.

Es decir, el árbol que hemos visto anteriormente no se genera al principio de una reparación y luego se recorre para probar cada uno de los reparadores posibles, sino que se va generando poco a poco y sólo cuando es necesario considerar una nueva alternativa.

8.2.4.3. Detección de violaciones relacionadas con los OIDs

Otra de las optimizaciones que originalmente no estaban contempladas en el método Queralt es la capacidad de detectar, tan pronto como se generan, hechos que van a ser rechazados tarde o temprano.

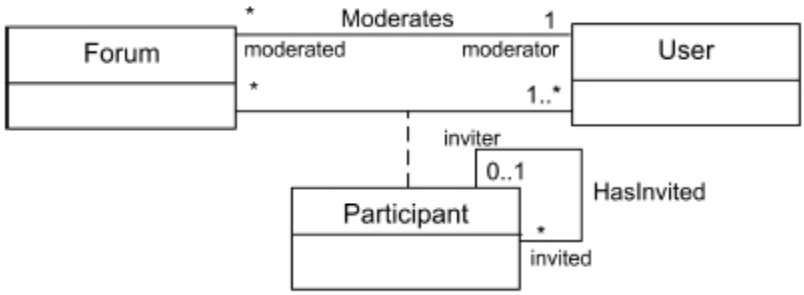
Por ejemplo, *Moderates(0,0)* es insatisfactible. No puede haber dos instancias diferentes (de Forum y User) con el mismo OID. En lugar de esperar a encontrar la restricción de integridad correspondiente, Aurus descarta el hecho en el momento de su construcción. Lo puede hacer porque que sabe que el primer término es de tipo Forum y el segundo de tipo User y sabe que no puede haber dos instancias diferentes con el mismo OID.

El algoritmo original detectaría la violación de la integridad del modelo tarde o temprano (al comprobar la restricción 0). La optimización es interesante porque se trata de una optimización sin apenas coste: se aprovechan estructuras de control ya existentes.

Me refiero a las estructuras necesarias para garantizar que no se van a repetir literales (User(5) y User(5), por ejemplo). No hay ninguna restricción de integridad que impida tal situación pero se debe impedir. Así pues, estas estructuras, que relacionan constantes con tipos o predicados, se pueden aprovechar para detectar inmediatamente violaciones relacionadas con los OIDs.

9. EJEMPLO

A continuación veremos una ejecución completa de Aurus para un modelo dado, el mismo que presentamos en capítulos anteriores:



context Forum inv:
--A user cannot participate in a forum he moderates
self.user -> select(u |u=self.moderator) -> size() = 0

context Participant inv:
--A participant cannot invite himself
self.inviter <> self

context Forum inv:
--A forum have at least one participant invited by another participant of the same forum
self.participant -> select(p | p.inviter.forum=self) -> size() > 0

En caso de no existir, Aurus crea una nueva carpeta output/fórum donde genera cuatro archivos de texto (logs) con información sobre la traducción, construcción del grafo, análisis del grafo y resultados de las *queries*. Hay que tener en cuenta que Aurus sobrescribiría directamente los archivos que se encontraran en la carpeta, en caso de que ésta ya existiera en el sistema de ficheros.

A continuación transcribo su contenido con mínimas modificaciones de formato.

forum_translation_log.txt

```
+ Classes and associations

  Forum(F)
  User(U)

  HasInvited(P1,P2)
  Moderates(F1,U2)
  Participant(P1,F2,U3)

+ Constraints for OIDs

  <- Forum(X) ^ User(X)
  <- Forum(X) ^ Participant(X,F1,U2)
  <- User(X) ^ Participant(X,F1,U2)

+ Constraints for hierarchies

+ Constraints for associations

  IsParticipant(P0) <- Participant(P0,F0,U0)
  <- HasInvited(P0,P1) ^ ¬IsParticipant(P0)
  <- HasInvited(P0,P1) ^ ¬IsParticipant(P1)
  <- Moderates(F0,U0) ^ ¬Forum(F0)
  <- Moderates(F0,U0) ^ ¬User(U0)
  <- Participant(OID,F0,U0) ^ ¬Forum(F0)
  <- Participant(OID,F0,U0) ^ ¬User(U0)
  <- Participant(OID1,F0,U0) ^ Participant(OID2,F0,U0) ^ OID1<>OID2

+ Constraints for association cardinalities

  MinModerator(F0) <- Moderates(F0,U0)
  MinUserForum(F0) <- Participant(P0,F0,U0)

  <- Forum(F0) ^ ¬MinModerator(F0)
  <- Forum(F0) ^ ¬MinUserForum(F0)

  <- HasInvited(P1,P0) ^ HasInvited(P2,P0) ^ P2<>P1
  <- Moderates(F0,U0) ^ Moderates(F0,U1) ^ U1<>U0

+ Constraints for attributes

+ Constraints for attribute cardinalities

+ OCL constraints

  <- Forum(F) ^ Participant(P0,F,U0) ^ Moderates(F,U0)
  <- Participant(P0,F0,U0) ^ HasInvited(P0,P0)
  <- Forum(F) ^ ¬AuxMain3(F)
  AuxMain3(F) <- Forum(F) ^ Participant(P0,F,U0) ^ HasInvited(P0,P1)
                 ^ Participant(P1,F,U1)
```

forum_graphConstruction_log.txt

```
ORIGINAL GRAPH NODES

0:      <- Forum(X) ^ User(X)
1:      <- Forum(X) ^ Participant(X,F1,U2)
2:      <- User(X) ^ Participant(X,F1,U2)
3:      <- HasInvited(P0,P1) ^ ¬IsParticipant(P0)
        IsParticipant(P0) <- Participant(P0,F0,U0)
4:      <- HasInvited(P0,P1) ^ ¬IsParticipant(P1)
        IsParticipant(P1) <- Participant(P1,F0,U0)
5:      <- HasInvited(P1,P0) ^ HasInvited(P2,P0) ^ P2<>P1
6:      <- Moderates(F0,U0) ^ ¬Forum(F0)
7:      <- Moderates(F0,U0) ^ ¬User(U0)
8:      <- Forum(F0) ^ ¬MinModerator(F0)
        MinModerator(F0) <- Moderates(F0,U0)
9:      <- Moderates(F0,U0) ^ Moderates(F0,U1) ^ U1<>U0
10:     <- Participant(OID,F0,U0) ^ ¬Forum(F0)
11:     <- Participant(OID,F0,U0) ^ ¬User(U0)
12:     <- Forum(F0) ^ ¬MinUserForum(F0)
        MinUserForum(F0) <- Participant(P0,F0,U0)
13:     <- Participant(OID1,F0,U0) ^ Participant(OID2,F0,U0) ^
            OID1<>OID2
14:     <- Forum(F) ^ Participant(P0,F,U0) ^ Moderates(F,U0)
15:     <- Participant(P0,F0,U0) ^ HasInvited(P0,P0)
16:     <- Forum(F) ^ ¬AuxMain3(F)
        AuxMain3(F) <- Forum(F) ^ Participant(P0,F,U0) ^
                       HasInvited(P0,P1) ^ Participant(P1,F,U1)


POTENTIAL VIOLATORS AND REPAIRS

  0's potential violators:   Forum, User
  0's repairs :

  1's potential violators:   Forum, Participant
  1's repairs :

  2's potential violators:   User, Participant
  2's repairs :

  3's potential violators:   HasInvited
  3's repairs :              Participant

  4's potential violators:   HasInvited
  4's repairs :              Participant

  5's potential violators:   HasInvited
  5's repairs :

  6's potential violators:   Moderates
  6's repairs :              Forum
```

```
  7's potential violators:   Moderates
  7's repairs :              User

  8's potential violators:   Forum
  8's repairs :              Moderates

  9's potential violators:   Moderates
  9's repairs :

  10's potential violators:  Participant
  10's repairs :             Forum

  11's potential violators:  Participant
  11's repairs :             User

  12's potential violators:  Forum
  12's repairs :             Participant

  13's potential violators:  Participant
  13's repairs :

  14's potential violators:  Forum, Participant, Moderates
  14's repairs :

  15's potential violators:  Participant, HasInvited
  15's repairs :

  16's potential violators:  Forum
  16's repairs :             Forum, Participant, HasInvited


ORIGINAL GRAPH ARCS

  Participant(3,1)
  Participant(3,2)
  Participant(3,10)
  Participant(3,11)
  Participant(3,13)
  Participant(3,14)
  Participant(3,15)
  Participant(4,1)
  Participant(4,2)
  Participant(4,10)
  Participant(4,11)
  Participant(4,13)
  Participant(4,14)
  Participant(4,15)
  Forum(6,0)
  Forum(6,1)
  Forum(6,8)
  Forum(6,12)
  Forum(6,14)
  Forum(6,16)
  User(7,0)
  User(7,2)
```

```
  Moderates(8,6)
  Moderates(8,7)
  Moderates(8,9)
  Moderates(8,14)
  Forum(10,0)
  Forum(10,1)
  Forum(10,8)
  Forum(10,12)
  Forum(10,14)
  Forum(10,16)
  User(11,0)
  User(11,2)
  Participant(12,1)
  Participant(12,2)
  Participant(12,10)
  Participant(12,11)
  Participant(12,13)
  Participant(12,14)
  Participant(12,15)
  Forum, Participant, HasInvited(16,0)
  Forum, Participant, HasInvited(16,1)
  Forum, Participant, HasInvited(16,1)
  Forum, Participant, HasInvited(16,2)
  Forum, Participant, HasInvited(16,3)
  Forum, Participant, HasInvited(16,4)
  Forum, Participant, HasInvited(16,5)
  Forum, Participant, HasInvited(16,8)
  Forum, Participant, HasInvited(16,10)
  Forum, Participant, HasInvited(16,11)
  Forum, Participant, HasInvited(16,12)
  Forum, Participant, HasInvited(16,13)
  Forum, Participant, HasInvited(16,14)
  Forum, Participant, HasInvited(16,14)
  Forum, Participant, HasInvited(16,15)
  Forum, Participant, HasInvited(16,15)
  Forum, Participant, HasInvited(16,16)


GRAPH LABELS

  A: Participant
  B: Forum
  C: User
  D: Moderates
  E: Forum, Participant, HasInvited
```

forum_graphAnalysis_log.txt

```
SUPERFLUOUS ARCS

  Forum(6,8)
  Moderates(8,6)
  Forum(10,12)
  Participant(12,10)


CYCLES

  Cycle -1:   Finite    ·Forum(10,16)·
                        ·Forum, Participant, HasInvited(16,3)·
                        ·Participant(3,10)·

  Cycle -2:   Finite    ·Forum(10,16)·
                        ·Forum, Participant, HasInvited(16,10)·

  Cycle -3:   Finite    ·Forum, Participant, HasInvited(16,16)·

  Cycle -4:   Finite    ·Forum(10,16)·
                        ·Forum, Participant, HasInvited(16,4)·
                        ·Participant(4,10)·


INCOMING DEGREES OF EACH NODE

  Degree    Nodes
  0:        6
  1:        7, 9, -1, -2, -3, -4
  2:
  3:
  4:        5
  5:        12
  6:
  7:        8, 11, 13
  8:
  9:        2
  10:       0
  11:       15
  12:
  13:
  14:
  15:       1
  16:       14
```

forum_validationResults.txt

```
EXECUTION RESULTS (queries and answers)

Query:      Moderates(0.0,1.0)
Answer:     Satisfiable
Facts:      Moderates(0.0,1.0) Forum(0.0) User(1.0)
            Participant(2.0,0.0,3.0) HasInvited(2.0,4.0)
            Participant(4.0,0.0,5.0) User(3.0) User(5.0)
Time:       6406 ms

Query:      Moderates(0.0,0.0)
Answer:     Unsatisfiable
Time:       344 ms

Query:      Moderates(F,U)
Answer:     Satisfiable
Facts:      Moderates(0.0,1.0) Forum(0.0) User(1.0)
            Participant(2.0,0.0,3.0) HasInvited(2.0,4.0)
            Participant(4.0,0.0,5.0) User(3.0) User(5.0)
Time:       5359 ms

Query:      HasInvited(1.0,1.0)
Answer:     Unsatisfiable
Time:       188 ms

Query:      Moderates(F,U) Participant(P,F,U)
Answer:     Unsatisfiable
Time:       182609 ms

Query:      Moderates(0.0,1.0) ¬Participant(X,0.0,Y)
Answer:     Unsatisfiable
Time:       375 ms

Query:      Moderates(0.0,1.0) ¬Participant(2.0,0.0,X)
Answer:     Satisfiable
Facts:      Moderates(0.0,1.0) Forum(0.0) User(1.0)
            Participant(3.0,0.0,4.0) HasInvited(3.0,5.0)
            Participant(5.0,0.0,2.0) User(4.0) User(2.0)
Time:       4125 ms

Query:      Moderates(F,U) ¬Participant(P,F,U)
Answer:     Satisfiable
Facts:      Moderates(0.0,1.0) Forum(0.0) User(1.0)
            Participant(2.0,0.0,3.0) HasInvited(2.0,4.0)
            Participant(4.0,0.0,5.0) User(3.0) User(5.0)
Time:       3781 ms

Query:      Participant(0.0,1.0,2.0) Participant(3.0,1.0,4.0)
            HasInvited(0.0,3.0)
Answer:     Satisfiable
Facts:      Participant(0.0,1.0,2.0) Participant(3.0,1.0,4.0)
            HasInvited(0.0,3.0) Forum(1.0) Moderates(1.0,5.0)
            User(5.0) User(2.0) User(4.0)
Time:       344 ms
```

10. CONCLUSIONES Y PROBLEMAS ABIERTOS

10.1. Sobre la herramienta Aurus

En mi opinión, después de varios meses de trabajo, puedo decir que el resultado ha sido satisfactorio. Se ha conseguido implementar una herramienta que respeta los requisitos de funcionalidad originales: Aurus es capaz de responder preguntas sobre la satisfactibilidad o insatisfactibilidad de ciertas propiedades en un modelo dado, así como realizar un análisis previo acerca de la decidibilidad o indecidibilidad del modelo.

No hay que olvidar, no obstante, que se trata de un prototipo todavía en una fase *beta*. Es decir, todavía es necesario realizar nuevas pruebas, solucionar posibles *bugs* y perfeccionar la estabilidad del sistema.

Dado que en este proyecto nos hemos centrado en desarrollar el núcleo de Aurus, hay un conjunto de funcionalidades y características que quedaron descartadas. En concreto, se decidió aplazar la implementación de una interfaz gráfica y de un módulo parseador-simplificador de OCL. Sin embargo, se espera poder incluir tales extensiones en una nueva versión de Aurus en el futuro.

10.2. Sobre el coste real del desarrollo de la herramienta

Principalmente, se aprecian dos diferencias entre la primera versión de la planificación y la evolución real del desarrollo del proyecto: la envergadura del proyecto y la sincronía/asincronía entre las tareas.

En lo que respecta a la envergadura, está claro que al principio fui demasiado optimista y el coste temporal del proyecto parecía menor del que realmente fue.

Por una parte, en lugar de las 500 horas que estaban reservadas para el desarrollo de Aurus, fueron necesarias unas 650. La razón de esta desviación en los pronósticos se encuentra en las dificultades que se escondían tras el diseño e implementación de la mayoría de componentes, destacando la validación del modelo. Sobre el papel, en la teoría del método Queralt, todo parecía más fácil y rápido. Pero a menudo, tras una simple frase que ocupa apenas una línea se ocultan horas de trabajo.

Tareas	Inicio	Fin	Horas
Cargando un modelo	1-4-08	18-1-09	164
Estrategia	1-4-08	10-4-08	10
Implementación	8-4-08	9-4-08	7
Creación ejemplos	10-4-08	14-1-09	30
Pruebas	15-4-08	19-1-09	15
Traducción UML	5-5-08	22-7-08	102
Estrategia	5-5-08	19-5-08	10
Especificación	7-5-08	11-5-08	3
Diseño	12-5-08	17-5-08	4
Implementación	15-5-08	23-7-08	80
Pruebas	25-6-08	23-7-08	5
Traducción OCL	20-7-08	22-9-08	158
Estrategia	20-7-08	5-8-08	13
Especificación	26-7-08	30-7-08	2
Diseño	31-7-08	8-8-08	3
Implementación	5-8-08	22-9-08	130
Pruebas	20-8-08	23-9-08	10
Construcción grafo	10-9-08	19-10-08	49
Estrategia	10-9-08	26-9-08	8
Especificación	14-9-08	19-9-08	2
Diseño	21-9-08	23-9-08	3
Implementación	22-9-08	20-10-08	32
Pruebas	30-9-08	20-10-08	4

Tareas	Inicio	Fin	Horas
Análisis ciclos	15-10-08	14-11-08	54
Estrategia	15-10-08	24-10-08	9
Especificación	17-10-08	20-10-08	1
Diseño	21-10-08	27-10-08	3
Implementación	22-10-08	14-11-08	37
Pruebas	27-10-08	15-11-08	4
Validación esquema	28-10-08	12-1-09	132
Estrategia	28-10-08	13-11-08	12
Especificación	30-10-08	15-11-08	5
Diseño	16-11-08	22-12-08	8
Implementación	7-11-08	13-1-09	100
Pruebas	22-12-08	13-1-09	7
Documentación	29-10-08	29-1-09	95
Informe	29-10-08	4-11-08	5
Memoria	22-12-08	22-1-09	65
Presentación	23-1-09	30-1-09	25
			754

Por otra parte, no tuve en cuenta las influencias externas. Supuse que me dedicaría con más intensidad al proyecto de lo que inevitablemente me pude dedicar, debido a diversos factores, como el trabajo paralelo que otras asignaturas también requerían. Eso provocó que la planificación, por ser demasiado rígida, no pudiera absorber las horas extras incrementando la concentración del trabajo diario, por ejemplo. Por lo tanto, la fecha límite para concluir el desarrollo de la herramienta tuvo que aplazarse hasta mediados de enero.

Otra de las diferencias que se observa claramente es que mientras la planificación inicial está más ordenada y sus tareas están sincronizadas, la planificación final es más caótica. La planificación inicial se diseñó de manera que las etapas se concluyeran completamente y dieran paso a la siguiente. Sin embargo, la realidad fue que al estar diseñando la parte de la generación de grafos se ponía de manifiesto que la traducción contenía algún error. Los solapamientos entre tareas que se aprecian en los diagramas de Gantt finales hacen referencia a este tipo de situaciones.

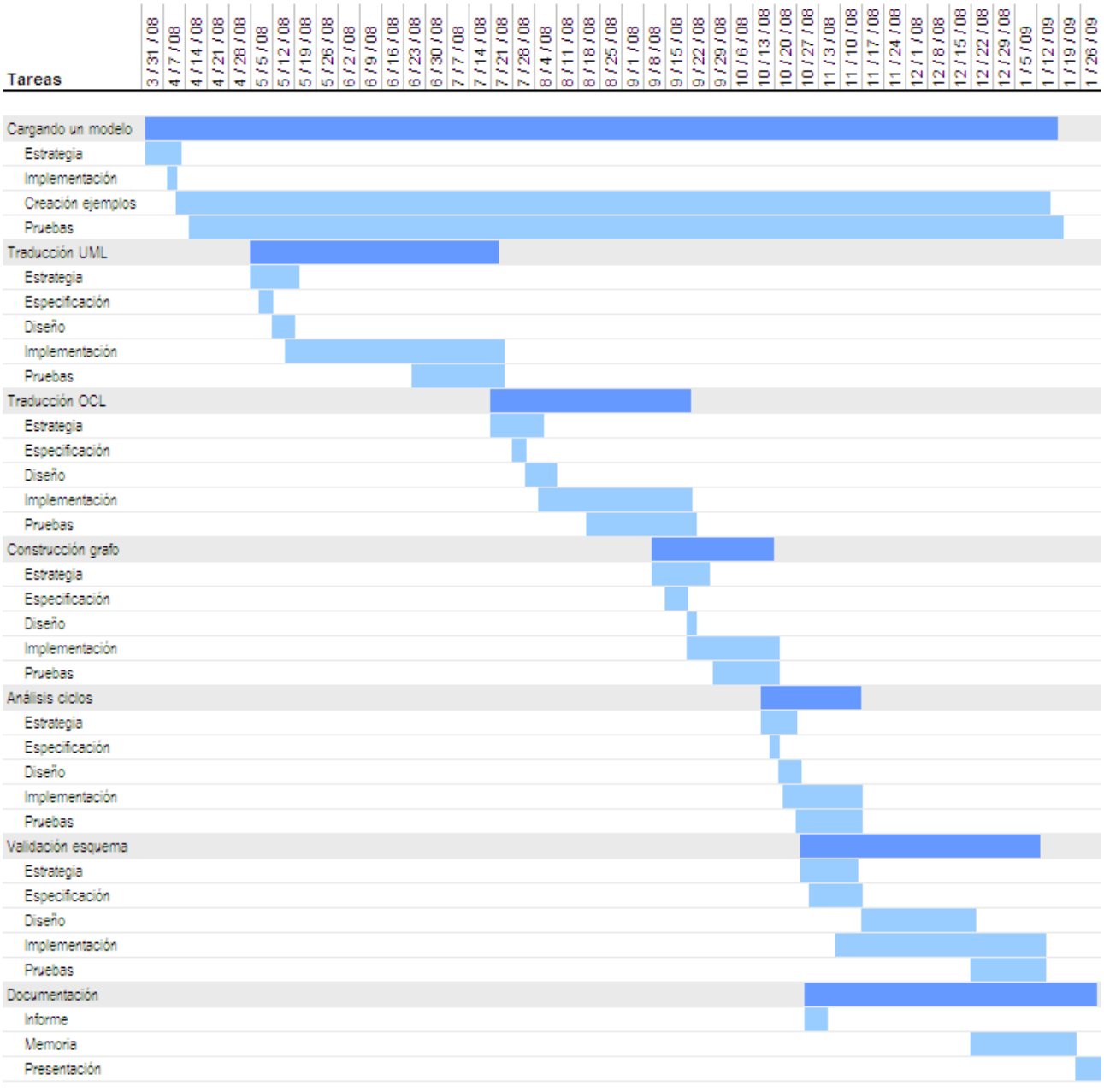


Figura 10: Diagrama de Gantt de la evolución real del proyecto

Solapando las tareas de análisis de ciclos y validación es como he intentado representar que había parte del análisis, concretamente la aplicación del teorema 3, que podía retroalimentarse con la validación. Como el tercer teorema sugiere una simulación de la validación para un ciclo dado, decidí realizar ambas tareas simultáneamente.

Adicionalmente, se observa que el tiempo que invertí en la construcción de ejemplos se alarga durante todo el proyecto. Esto es porque, a medida que iba concluyendo etapas, me daba cuenta de que los ejemplos que había preparado inicialmente no eran suficientes para testear las funcionalidades.

También podemos ver que el tiempo de implementación en esta fase es considerablemente inferior en la planificación inicial, ya que al principio se pensaba que sería suficiente con utilizar una herramienta de modelado para fabricar los modelos de ejemplo. Sin embargo, la mayoría de modelos necesitaban completarse ejecutando un pequeño código que había que implementar específicamente.

Por último, me gustaría aclarar que el número de días empleado en una tarea no tiene porqué ser proporcional al número de horas, ya que el número de horas al día que dedicaba al proyecto no era constante.

Así las cosas, sabiendo que el coste del desarrollo del proyecto ha sido de 650 horas y que la retribución media aproximada de un analista-programador es de 35 € / hora, podemos fijar el coste real del proyecto en 650 * 35 = 22.750 €.

10.3. Sobre el proyecto en general

Sin duda, durante la elaboración de este proyecto he aprendido muchas cosas. Entre ellas, he aumentado considerablemente mi destreza en programación orientada a objetos. También he aprendido una lección muy importante, y es que tras el concepto más sencillo se pueden ocultar horas y horas de trabajo. Esto es muy importante al analizar el coste de un proyecto: no se deben subestimar los detalles.

Además, he podido aplicar los conocimientos y las habilidades que durante estos cinco años he estado desarrollando. He podido poner en práctica el que creo que es el recurso más útil para un ingeniero informático hoy en día: la capacidad de adaptación. En un mundo donde la tecnología de ayer es diferente de la tecnología de mañana, aprender y adaptarse a cualquier entorno y cualquier metodología, y hacerlo rápido, es crucial para su supervivencia profesional.

Hace apenas ocho meses, no sabía nada acerca de la validación de esquemas conceptuales. A día de hoy he logrado implementar una herramienta que realiza esta tarea automáticamente.

12. BIBLIOGRAFÍA Y REFERENCIAS

[1] EinaGMC Project
Grupo de trabajo GMC (Modelització Conceptual) de la UPC y la UOC
Recurso electrónico: http://guifre.lsi.upc.edu/eina_GMC

[2] Reasoning on UML Class Diagrams with OCL Constraints
Anna Queralt, Ernest Teniente
International Conference on Conceptual Modeling - ER'06, LNCS vol. 4215 , pgs. 497-512
Springer Berlin / Heidelberg
Noviembre 2006

[3] Decidable Reasoning in UML Schemas with Constraints
Anna Queralt, Ernest Teniente
Conference on Advanced Information Systems Engineering, LNCS vol. 5074 , pgs. 281-295
Springer Berlin / Heidelberg
Junio 2008

[4] Checking query containment with the CQC method
Carles Farré, Ernest Teniente, Toni Urpí
Data & Knowledge Engineering, vol. 53, pgs 163-223
Elsevier Science Publishers B. V.
Mayo 2005

[5] Metamodelo UML2.0
OMG, 2007
Recurso electrónico: http://www.omg.org/docs/formal/07-11-04.pdf

[6] Metamodelo OCL2.0
OMG, 2008
Recurso electrónico: http://www.omg.org/docs/ptc/03-10-14.pdf

[7] Especificació de sistemes software en UML
Dolors Costal; María Ribera Sancho; Ernest Teniente
Edicions UPC, 2003

[8] Validación de esquemas de bases de datos SQL Server
Carlos Beltrán; director: Carles Farré
Projecte Final de Carrera, FIB-UPC
2003

[9] Processador d'expressions OCL en un entorn de modelització conceptual
Antonio Villegas; director: Antoni Olivé
Projecte de Final de Carrera, FIB-UPC
Junio 2008

[10] Eclipse Modeling Framework Project (EMF)
Entorno de trabajo de modelización y generación de código
Recurso electrónico: http://www.eclipse.org/modeling/emf/

[11] Start Guide Core EinaGMC (EinaGMC Project)
Albert Tort
http://guifre.lsi.upc.edu/eina_GMC/resources_eina/core/start_guide_core.pdf

[12] Documentació d'usuari - XMIConverter (EinaGMC Project)
Elena Planas
http://guifre.lsi.upc.edu/eina_GMC/resources_eina/XMIConverter/XMIConverter_guia_usuari.pdf

[13] JGraphT
Librería para la implementación de diferentes tipos de grafos
Recurso electrónico: http://jgrapht.sourceforge.net

[14] Manual JGraph
Librería para la visualización de diferentes tipos de grafos
Recurso electrónico: http://www.jgraph.com/pub/jgraphpadmanual.pdf

13. ANEXO: EJEMPLO DE CONVERSIÓN UML - XMI

Para que Aurus sea capaz de razonar sobre un modelo UML se debe disponer de una versión en formato XMI de éste. En el capítulo 4.3 se explica el procedimiento para obtener este documento XMI, que a continuación quedará ilustrado a partir de un ejemplo.

El punto de partida es el propio modelo UML. Por ejemplo:

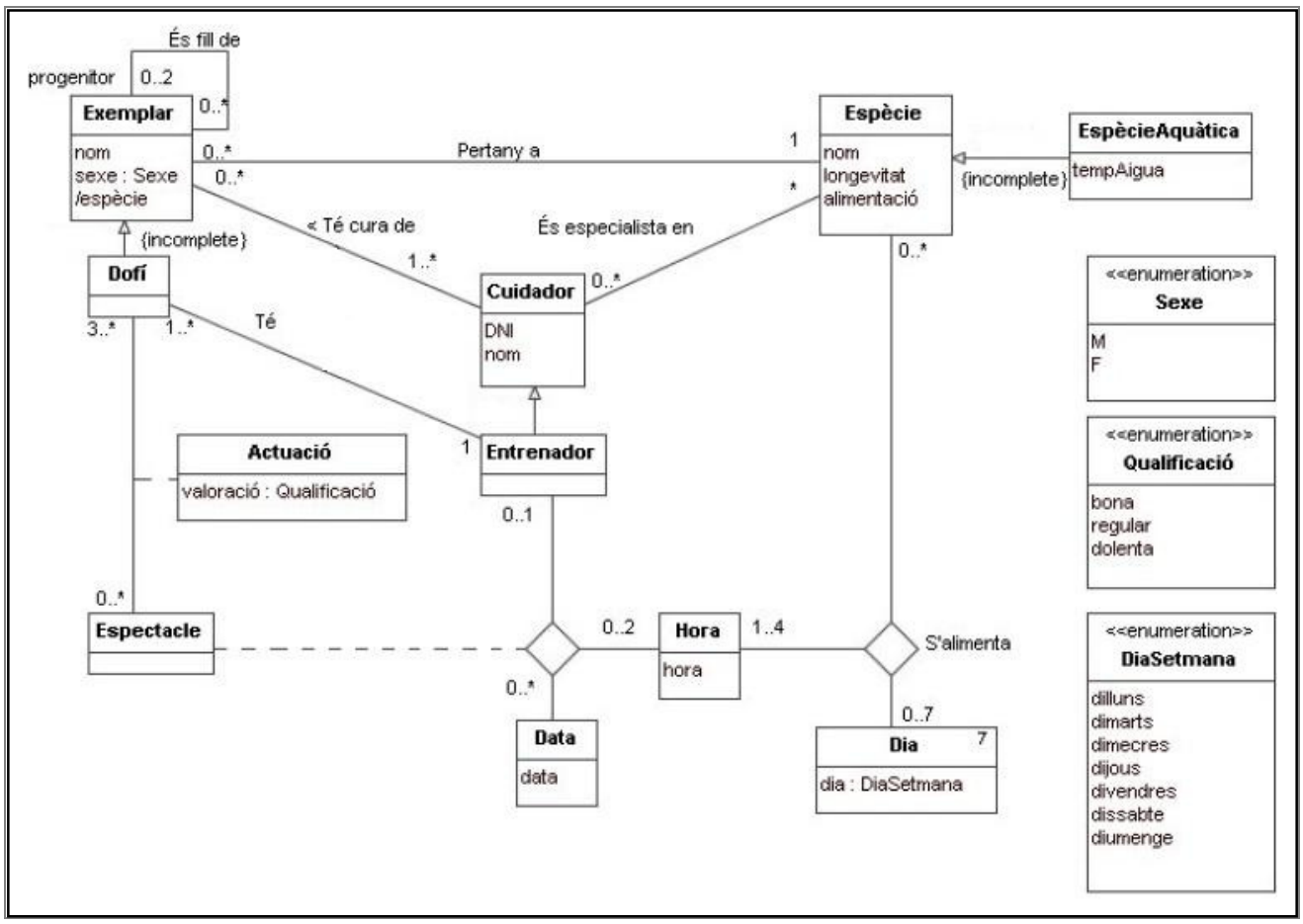


Figura 11: Ejemplo de modelo conceptual

En primer lugar, utilizamos una herramienta de modelado (Poseidon, en nuestro caso) para diseñar tantos fragmentos del esquema como sea posible. Para el modelo de ejemplo se consiguen diseñar las partes que se aprecian en la figura 12.

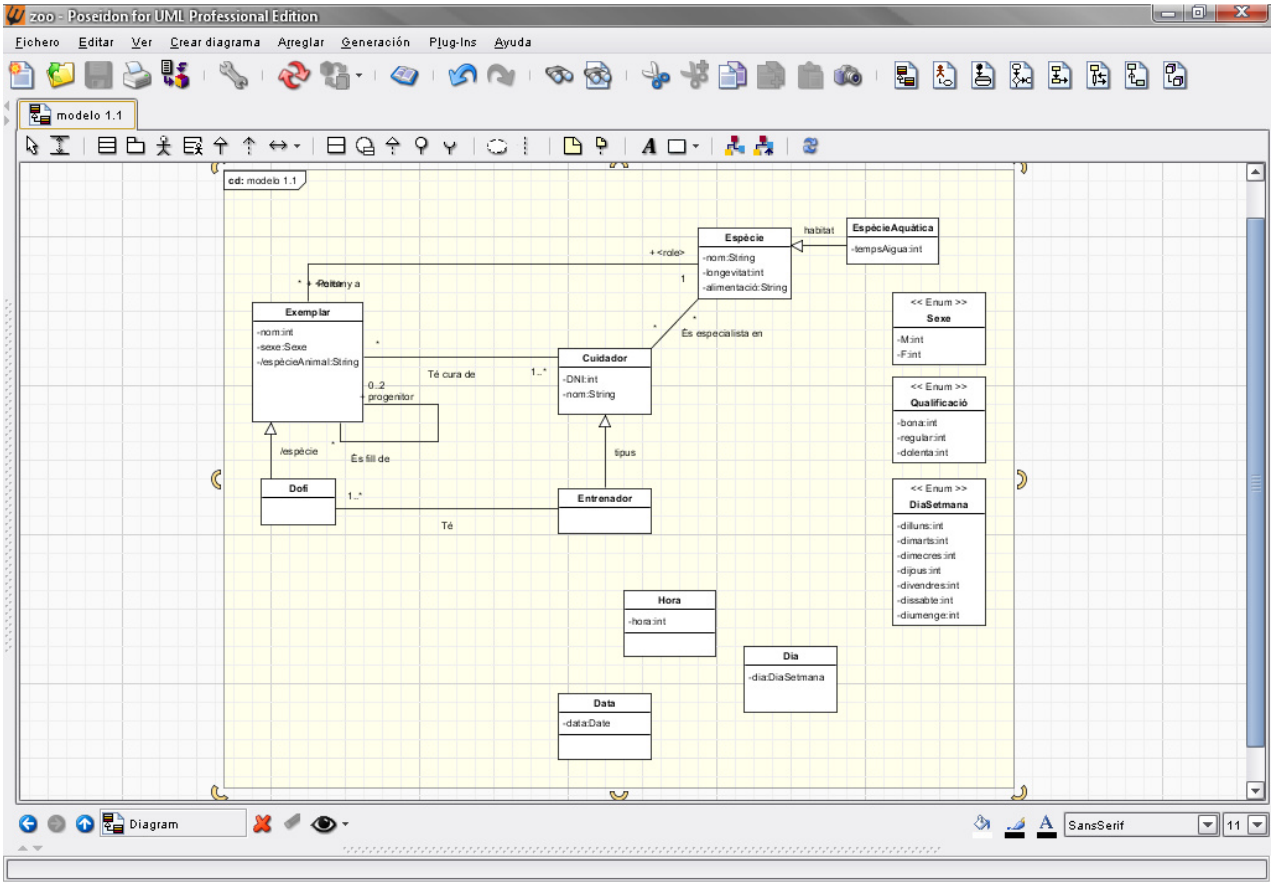


Figura 12: Captura de pantalla de Poseidon

Una vez acabamos de modelar con Poseidon, exportamos a XMI. A continuación hemos de convertir este XMI de Poseidon a XMI de EinaGMC. Para ello necesitamos invocar operaciones de la API XMIConverter, concretamente las siguientes líneas de código:

```
import XMIConverter.ConverterToEinaGMC;
ConverterToEinaGMC cte = new ConverterToEinaGMC();
cte.convert(input, output, "Poseidon");
```

Ahora tenemos nuestro document XMI en formato EinaGMC, tal como queremos. Sólo falta completarlo, para lo que invocaremos operaciones de la API del core de EinaGMC.

En las páginas siguientes se muestra el código completo de la función requerida.

```
private static void zoo(String output) throws Exception
{
      Project pro = new Project("zoo");
      pro.importXMI(output);

      UserFacade uf = new UserFacade(pro);

      // We transform classes which are enumerations in enumerations proper

      LinkedList<String> enums = new LinkedList<String>();
      enums.add("Sexe");
      enums.add("Qualificació");
      enums.add("DiaSetmana");
      changeIntoEnumerations(pro,uf,enums);

      // We create the association "S'alimenta"

      UmlClass especie, hora, dia;
      especie = uf.findClassByName("Espècie");
      hora = uf.findClassByName("Hora");
      dia = uf.findClassByName("Dia");

      LinkedList<Property> associationEnds = new LinkedList<Property>();
      PropertyFacade propf = new PropertyFacade(pro);
      LiteralIntegerFacade lif = new LiteralIntegerFacade(pro);
      LiteralInteger lilower, liupper;

      Property pDia = propf.createProperty(false, false, "",
      uml2.kernel.VisibilityKindEnum.PUBLIC, false, false, false,
      uml2.kernel.AggregationKindEnum.NONE, false, false);
      pDia.setType(dia);
      lilower = lif.createLiteralInteger();
      lilower.setValue(0);
      liupper = lif.createLiteralInteger();
      liupper.setValue(7);
      pDia.setLowerValue(lilower);
      pDia.setUpperValue(liupper);

      Property pEspecie = propf.createProperty(false, false, "",
      uml2.kernel.VisibilityKindEnum.PUBLIC, false, false, false,
      uml2.kernel.AggregationKindEnum.NONE, false, false);
      pEspecie.setType(especie);
      lilower = lif.createLiteralInteger();
      lilower.setValue(0);
      liupper = lif.createLiteralInteger();
      liupper.setValue(-1);
      pEspecie.setLowerValue(lilower);
      pEspecie.setUpperValue(liupper);

      Property pHora = propf.createProperty(false, false, "",
      uml2.kernel.VisibilityKindEnum.PUBLIC, false, false, false,
      uml2.kernel.AggregationKindEnum.NONE, false, false);
      pHora.setType(hora);
      lilower = lif.createLiteralInteger();
      lilower.setValue(1);
      liupper = lif.createLiteralInteger();
      liupper.setValue(4);
      pHora.setLowerValue(lilower);
```

```
pHora.setUpperValue(liupper);

associationEnds.add(pEspecie);
associationEnds.add(pDia);
associationEnds.add(pHora);

AssociationFacade assf = new AssociationFacade(pro);
Association a = assf.createAssociation(associationEnds);
a.setName("S'alimenta");


// We create the association "Espectacle"

UmlClass entrenador, data, hora2;
entrenador = uf.findClassByName("Entrenador");
data = uf.findClassByName("Data");
hora2 = uf.findClassByName("Hora");

associationEnds = new LinkedList<Property>();

Property pEntrenador = propf.createProperty(false, false, "",
uml2.kernel.VisibilityKindEnum.PUBLIC, false, false, false,
uml2.kernel.AggregationKindEnum.NONE, false, false);
pEntrenador.setType(entrenador);
lilower = lif.createLiteralInteger();
lilower.setValue(0);
liupper = lif.createLiteralInteger();
liupper.setValue(1);
pEntrenador.setLowerValue(lilower);
pEntrenador.setUpperValue(liupper);

Property pData = propf.createProperty(false, false, "",
uml2.kernel.VisibilityKindEnum.PUBLIC, false, false, false,
uml2.kernel.AggregationKindEnum.NONE, false, false);
pData.setType(data);
lilower = lif.createLiteralInteger();
lilower.setValue(0);
liupper = lif.createLiteralInteger();
liupper.setValue(-1);
pData.setLowerValue(lilower);
pData.setUpperValue(liupper);

Property pHora2 = propf.createProperty(false, false, "",
uml2.kernel.VisibilityKindEnum.PUBLIC, false, false, false,
uml2.kernel.AggregationKindEnum.NONE, false, false);
pHora2.setType(hora2);
lilower = lif.createLiteralInteger();
lilower.setValue(0);
liupper = lif.createLiteralInteger();
liupper.setValue(2);
pHora2.setLowerValue(lilower);
pHora2.setUpperValue(liupper);

associationEnds.add(pEntrenador);
associationEnds.add(pHora2);
associationEnds.add(pData);

AssociationClassFacade asscf = new AssociationClassFacade(pro);
AssociationClass ac = asscf.createAssociationClass(associationEnds);
```

```
        ac.setName("Espectacle");

        // We create the association "Actuació"

        UmlClass dofi;
        AssociationClass espectacle = ac;
        dofi = uf.findClassByName("Dofí");

        associationEnds = new LinkedList<Property>();

        Property pEspectacle = propf.createProperty(false, false, "",
        uml2.kernel.VisibilityKindEnum.PUBLIC, false, false, false,
        uml2.kernel.AggregationKindEnum.NONE, false, false);
        pEspectacle.setType(dofi);
        lilower = lif.createLiteralInteger();
        lilower.setValue(3);
        liupper = lif.createLiteralInteger();
        liupper.setValue(-1);
        pEspectacle.setLowerValue(lilower);
        pEspectacle.setUpperValue(liupper);

        Property pDofi = propf.createProperty(false, false, "",
        uml2.kernel.VisibilityKindEnum.PUBLIC, false, false, false,
        uml2.kernel.AggregationKindEnum.NONE, false, false);
        pDofi.setType(espectacle);
        lilower = lif.createLiteralInteger();
        lilower.setValue(0);
        liupper = lif.createLiteralInteger();
        liupper.setValue(-1);
        pDofi.setLowerValue(lilower);
        pDofi.setUpperValue(liupper);

        associationEnds.add(pEspectacle);
        associationEnds.add(pDofi);

        AssociationClass ac2 = asscf.createAssociationClass(associationEnds);
        ac2.setName("Actuació");

        Property valoracio = propf.createProperty(false, false, "",
        uml2.kernel.VisibilityKindEnum.PUBLIC, false, false, false,
        uml2.kernel.AggregationKindEnum.NONE, false, false);
        valoracio.setName("valoració");
        DataType qualificacio = (DataType)uf.findDataTypeByName("Qualificació");
        valoracio.setType((DataType)qualificacio);
        valoracio.setUmlclass(ac2);

        tuneSpelling(pro,uf);
        setDataTypes(pro,uf);

        pro.saveXMI(output.substring(0, output.length()-4)+"_completed.xmi");
        pro.closeProject();
}
```